Thesis Proposal

Never-Ending Learning for Open-Domain Semantic Parsing

Author:
Abulhair Saparov

Advisor:
Tom Mitchell

Abstract

In this thesis, we explore the hypothesis that techniques from never-ending learning and curriculum learning can be applied to the problem of training an open-domain semantic parser. To train such a parser in a traditional supervised setting would require an insurmountable amount of labeled data. Instead, we propose a method that uses a semantic parser to learn definitions of new previously-unrecognized words, and to induce and populate a first-order theory that explains (via logical entailment) the observed sentences. As definitions of new words are parsed, they can be used to produce self-supervised training examples for the semantic parser, increasing its vocabulary of recognized words. Our parser outputs partial logical forms for sentences that contain unrecognized tokens. Therefore, if a sentence defines a previously unseen word, the unrecognized token can be linked to a new or existing concept in the theory, and the resulting logical form along with the sentence become a newly labeled training example. We apply Bayesian statistical machine learning, enabling every component in our proposed method to be interpretable, composeable with other models, and guided by prior information. For example, we can impose a prior that favors simpler and more consistent first-order theories over more complex ones. This also enables our semantic parser to be driven and guided by the background knowledge stored in the first-order theory, thus closing the loop and enabling the reading competence of the semantic parser to increase over time, as it reads more and more definitional sentences. We also propose a method to perform active data collection: to find a sequence of sentences to be read by the semantic parser such that its reading competence improves.

While this goal in its most general form is ambitious, we simplify the problem as much as possible without trivializing the interesting research questions. We focus on declarative, factual sentences, with little grammatical complexity, such as those found on Simple English Wikipedia. To evaluate our method, we can inspect the coverage of the learned first-order theory, inspect the size and quality of the learned vocabulary, and use a question-answering task, without any explicit (domain-specific) training data.

1. Introduction

Much of the work in artificial intelligence and machine learning focuses on developing methods and algorithms to solve domain-specific problems, oftentimes requiring supervised training examples. For example, in natural language processing, semantic parsers are largely trained for specific domains, on datasets where each sentence is labeled with a full logical form. A long-standing goal in these fields is to develop algorithms that can be applied across domains, and that perform well across many different tasks. Acquiring supervised training data is infeasible to train machine learning algorithms in open-domain settings, since there would need to be a set of training examples for a representative sample of multiple domains, and so the amount of required labeling is insurmountable. As a result, to date, there are no truly open-domain semantic parsers.

Never-ending learning is a machine learning paradigm in which an algorithm learns from examples continuously over time, in a largely self-supervised fashion, where its experience from past examples can be leveraged to learn future examples [22]. We attempt to train an open-domain semantic parser by applying never-ending learning where a semantic parser is trained to extract logical forms from a collection of *open-domain documents*, such as webpages on the internet, and populate a knowledge base. In this approach, the parser is initially only able to understand a small fraction of the sentences, but progressively improves its reading competence by carefully selecting sentences to read that define new words but are still understandable by the semantic parser with high confidence. As a result, such an approach can iteratively produce self-supervised training examples to further train the semantic parser. This method can be used to automatically produce a large corpus of self-labeled training examples, which would prove instrumental in many other problems in natural language processing and machine learning. Our approach requires a semantic parsing model that is aware of background knowledge and knowledge extracted from previous sentences. As such, we develop a generative model in which the logical form is generated from a prior distribution, and a random process converts this logical form into the output sentence. During parsing, given a sentence, we aim to maximize the posterior probability of the logical form. We can direct the semantic parsing optimization by placing higher prior probability on logical forms consistent with background knowledge. Furthermore, we develop a model of a knowledge base which we couple with our semantic parsing model, enabling our parser to extract rich logical forms from sentences and populate the knowledge base, jointly.

This approach employs semantic parsing, i.e. it extracts meaning from individual sentences within each document, which we term *micro-reading*. This is in contrast to *macro-reading* which involves extracting relations by gathering word frequency information across large numbers of documents, largely ignoring grammatical structure of individual sentences. We rely on micro-reading as opposed to macro-reading to be able to extract definitions of new words from individual sentences, choosing to read a smaller number of higher-quality lower-noise sentences rather than a larger number of possibly noisier sentences.

Thus, we aim to test the following hypothesis in this thesis:

> "*A knowledge-driven micro-reader can be trained to understand individual sentences in documents in the open domain, use knowledge about the world to guide semantic parsing and resolve ambiguous interpretations, compile and reason over the collection of acquired logical forms in a knowledge base, and find a sequence of sentences/documents to read such that its reading competence improves with self-supervision, with the goal of maximizing performance with respect to an evaluation metric, such as a question-answering task or a metric measuring the coverage of the knowledge base.*"

Too much fudge! Just say exactly what your simplificati ons are, instead of "for instance".

There are numerous research questions in this area of study. How is *reasoning* incorporated into the prior of our semantic parsing model? How do we ensure that the learned knowledge is sufficiently consistent? What is the quality of the self-supervised training examples? What is the appropriate set of seed training examples? How should the micro-reading be directed in order to maximize evaluation performance? We wish to explore many of these interesting research questions without being hampered by unnecessary complexity, and as a result, we will make many design choices in favor of simplicity. For instance, we restrict our method to read only factual information from documents on the internet, such as Wikipedia and Wiktionary. The knowledge base will consist of a collection of logical forms in a formal language. As the system reads new sentences, it will try to modify the knowledge base such that the logical forms in the knowledge base *logically entail* the observed sentences. As another example, the vast majority of sentences in natural language depend heavily on context, containing anaphora or pronouns that refer to objects in previous sentences. Tackling the problem of context modeling is beyond the scope of this thesis, but it may be necessary for a parser to be able to properly understand sentences in context to achieve broad coverage. If the parser's coverage is not sufficiently broad, it may not be possible to find a sequence of sentences/documents that the parser can read to improve its own competence. However, since the set of sentences is very large, for example on the internet, we may still be able to find high-quality

definitional sentences that don't heavily rely on context, such as online dictionaries. In this document, we first detail related work in section 2. The overview of our model and inference approach is described in section 3 with preliminary results in 3.1 and 3.2. We detail the proposed work in section 4.

2. Related work

Never-ending learning is closely related to lifelong learning [7, 35]. Previous work has applied never-ending learning to the problem of classifying noun phrases [22] and images [6] into one of a number of semantic categories, extracting relations from text, recognizing multi-word phrases [28], controlling robot behavior [31], etc. However, past applications of never-ending learning to problems in natural language processing relied on computing occurrence frequencies of words within the context of surrounding words across a very large number of documents, discarding any syntactic and pragmatic information in the process. Humans, on the other hand, can extract much more information from individual sentences, exploiting a large set of background knowledge and contextual information, and our approach attempts to capture this by using never-ending learning to train a semantic parser.

Existing semantic parsers are quite successful in particular domains [39, 40, 38, 18, 19, 20, 21, 41, 12]. However, they are largely domain-specific, relying on domain-specific supervision to achieve better performance. Although there has been some work towards open-domain semantic parsing [5, 20, 33], this goal remains elusive.

The work most related to our reasoning module, as we will describe below, is *inductive logic programming* (ILP) [23]. Given a set of observed logical forms, we wish to find a set of hypothetical logical forms (which we call the *theory*) that logically entails the observations. However, there are many possible theories that can explain a given collection of observed logical forms, and there does not always exist a single "correct" theory, especially if the observations express information about uncertainty. Probabilistic ILP approaches [24, 10, 30, 11] instead define a distribution over theories, and they typically search for the theory that maximizes the likelihood of the observed logical forms. However, we place a prior distribution over theories, favoring simpler theories over complex ones, where relations are very sparse, and types are structured according to an ontology. In addition, ILP typically restricts the logical formalism to Horn clauses, for tractable inference, whereas there are many natural language sentences that express semantics beyond what is representable by Horn clauses.

Some of our ideas are conceptually similar to [32], which jointly learns to parse sentences into logical forms and to classify objects as instances of concepts (such as whether an email is spam or not). Our work aims to learn semantically richer concepts, represented as expressions in a formal language, possibly with complex logical relationships to other concepts. In this setting, we can logically deduce whether an object is an instance of a concept, rather than relying on a separate classifier with its own set of assumptions.

3. Approach

Our never-ending micro-reading system consists of three primary components:

- **Language module:** A semantic parsing model that can convert natural language sentences into logical forms in a formal language.
- **Reasoning module:** The component containing the ontology and knowledge base. Given a set of observed logical forms, this component's task is to find a knowledge base and ontology that logically entail the observations, while preferring simpler and more compact descriptions of the world (i.e. logical induction).
- **Executive module:** This component determines which documents/sentences to read at any given time, with the goal of maximizing some evaluation metric, such as performance on an evaluation task or a metric measuring the coverage of the knowledge base.

We take the approach of representing knowledge symbolically as logical forms in a formal language. The ontology and knowledge base are modeled as a single *theory*, which is a collection of logical forms. The *language* module and *reasoning* module are modeled probabilistically (i.e. as a generative process). We describe the probabilistic model at a high-level in this section below. Further details for the *language* model are provided in section 3.2 as preliminary results. Details for the *reasoning* module are provided in section 4.1 as proposed work. Our description of these models will be agnostic with respect to the formal language. As such, the system can be feasibly built using a very large variety of formal languages. However, we propose a handful of semantic representations in section 4.2 each making a different trade-off between expressivity and tractability of reasoning and implementation. Finally, the *executive* module is also detailed as proposed work in section 4.3.

In order to perform reasoning over a formal language, we rely on a *deductive system*, which is a set of *deduction rules* $\mathcal{D} \triangleq \{d_1, \ldots, d_k\}$. Each deduction rule $d \in \mathcal{D}$ can be applied to zero or more premises and produce a conclusion. We say the conclusion is *logically entailed* by those premises. Given the premises and the deduction rule, the conclusion is uniquely determined. A *proof* π under the deductive system $\mathcal{D}$ is a sequence of logical forms $(\phi_1, \ldots, \phi_n)$, where each logical form in this sequence ϕ_i is labeled

with a deduction rule $d_i \in \mathcal{D}$. The premises of d_i must either be axioms or appear earlier in the proof. If there exists a proof of a logical form ϕ given a set of axioms $a_1, \ldots, a_r$, we write $a_1, \ldots, a_r \vdash \phi$.

The theory K that constitutes the knowledge base and ontology is a collection of logical forms $K \triangleq \{\phi_1, \ldots, \phi_n\}$. The theory K is called *consistent* if falsity cannot be proved: $K \nvdash \bot$. However, for sufficiently rich formal languages and deductive systems, it is computationally intractable to require K to be consistent. Thus, we instead require weaker notions of consistency. One option is to restrict the complexity of proofs of $K \vdash \bot$. For example, we can place a limit on the time given to a theorem prover to find the proof, or a maximum number of steps in the proof. We may not even need to check for consistency at *every* step. Instead, consistency checking can occur at every n steps, or at every n seconds. Going even further, we can avoid consistency checking altogether until the semantic parser reads a fact x that is a contradiction with respect to K (i.e. $K \vdash \neg x$).

Variable	**Description**
K	The theory, which is a set of logical forms in a formal language $\mathcal{L}_1$.
$\boldsymbol{\pi} \triangleq \{\pi_1, \ldots, \pi_n\}$	The proofs, where π_i is the proof of $K \vdash x_i$. The proofs rely on a fixed deductive system $\mathcal{D}$ for the formal language $\mathcal{L}_1$.
$\boldsymbol{x} \triangleq \{x_1, \ldots, x_n\}$	The logical forms in a formal language $\mathcal{L}_2$ that encode the meaning of each sentence.
$\boldsymbol{t} \triangleq \{t_1, \ldots, t_n\}$	The derivation trees (syntax trees) of each sentence.
$\boldsymbol{y} \triangleq \{y_1, \ldots, y_n\}$	The output sentences which are observed.

TABLE 3.1: List of variables in our model.

In order to apply statistical machine learning tools to this problem, we need to define a distribution over the variables in our model, which we list in figure 3.1. As such, we define a generative process over these variables, ultimately generating the observed sentences $\boldsymbol{y}$. The generative process first generates the K from a prior distribution $p(K)$. Note that we can impose further structure on K, such as greater regularity, an ontology, or axioms expressing subsumption, mutual exclusion, etc. In fact, regularity is essential for our system to learn how to generalize. We discuss a number of further assumptions that can be made about the structure of K in section 4.1. We will initially design our system to read only factual data, and so we assume that for every input sentence y_i, its logical form representation x_i (i.e. its semantic parse) is logically entailed by the theory: $K \vdash x_i$ for all i. Thus, for each sentence i, the generative process first generates a proof π_i of a logical form x_i, with the axioms given in K. Given the logical form x_i, the language module then generates the output sentence y_i. The language module implements the semantic grammar model and inference as described in [29]. The reasoning module governs the variables K, $\boldsymbol{\pi}$, and $\boldsymbol{x}$, whereas the language module governs $\boldsymbol{x}$, $\boldsymbol{t}$, and $\boldsymbol{y}$. Notice that $\mathcal{L}_1$ and $\mathcal{L}_2$ do not have to be identical. It is sufficient to define a function $f : \mathcal{L}_2 \mapsto \mathcal{L}_1$ that maps from logical forms in $\mathcal{L}_2$ to logical forms in $\mathcal{L}_1$. In this case, each π_i is a proof of $K \vdash f(x_i)$.

Although neural methods and non-symbolic representation learning are highly popular in natural language processing, currently, we choose to use statistical methods that operate over symbolic representations. Our preliminary work on the language module is built using a generative model where logical forms are represented symbolically. Rewriting this module in a non-symbolic framework would take unnecessary time and is beyond the scope of this thesis. In addition, interpretability will be critically important to diagnose errors in the model and to understand exactly what the modules are doing.

We propose a handful of formal languages for the language module in section 4.2, and as we note there, natural language utterances are very semantically rich. They can easily contain universal quantifiers, existential quantifiers, negation, and even higher-order structures. However, we can use neo-Davidsonian semantics to express higher-order structure in a first-order language [25]. Because of this, we choose first-order logic as the formal language in the reasoning module. We select a well-known deductive system for (classical) first-order logic known as *natural deduction* [15, 16].

The existence of natural language utterances that express uncertainty implies that K itself must contain information about uncertainty. Take as an example the sentence "The cat is probably not sleeping on the bed." We take the approach that K itself describes a single deterministic world, in which the cat is either sleeping or not. Given a set of observations $\boldsymbol{y} \triangleq \{y_i\}$, we can capture this uncertainty in the posterior distribution $p(K|\boldsymbol{y})$. This distribution will contain information about uncertain facts in the world, and so if we wish to generate sentences that express uncertainty, the above generative process would need to generate logical forms x_i from the posterior $p(K|\boldsymbol{y})$ rather than from K directly. This observation does support the use of an inference method that doesn't discard the probabilistic information contained in $p(K|\boldsymbol{y})$. Thus, algorithms that produce point-estimates of $p(K|\boldsymbol{y})$, such as expectation maximization, are inadequate. Rather, it is desirable to use inference that keeps more information about the posterior. As a result, we rely on Markov chain Monte Carlo (MCMC) [14, 27]. Thus, all uncertainty about K is encoded in the posterior representation $p(K|\boldsymbol{y})$ rather than, for example, as confidence values associated with each logical form in K. In contrast, the logical forms for each sentence x_i can contain information about uncertainty, for example by using a predicate such as `is_likely`.

To understand inference in this model, it is illuminating to consider the reverse of the generative process: In the language module, given a sentence y_i and a theory K, we have developed an algorithm that finds the logical form x_i that maximizes the posterior probability. In the reasoning module, for reasons mentioned above, we use MCMC to produce samples from the posterior of the theory K and the proofs $\boldsymbol{\pi}$. The overlying executive module determines the order in which sentences are read. It also creates self-supervised training examples for the language module.

We first consider a simple task to evaluate our system: the system is given a set of natural language questions and an amount of time during which the system is expected to "study" and prepare to answer the given questions. The executive module is aware of this, and is responsible for directing the behavior of the system to perform as well as possible on the evaluation.

Figure 3.1 displays an example of the end-to-end execution of our methodology. The various modules are first initialized: the reasoning module may be initialized to a previously learned knowledge base and ontology, or it may be initialized with a seed set of beliefs. Once initialization is complete, the main training loop begins: the executive module selects a sentence to read, taking into account the time remaining, the extent or lack of knowledge about various concepts, and the evaluation task. The language module (the semantic parser) then attempts to parse the given sentence. If there are any unrecognized words, our semantic parser outputs an incomplete parse, containing unknown concepts or predicates. For example, in iteration 1 in the figure, the noun "Pennsylvania" is unknown, and the parser output a logical form containing an unknown concept $?_1$. If the sentence is *definitional*, where it provides information about a previously unseen concept with sufficiently high confidence, the executive module chooses to add it as a self-supervised training example. If the parse does not have sufficiently high confidence, the executive module directs the parser to read definitional sentences for the unrecognized words, as in iteration 2 in the figure example. Once the parser outputs a logical form without unknown concepts or predicates, the reasoning module then attempts to incorporate the new logical form x^* into the theory K. It does so by using MCMC to estimate $p(K, \boldsymbol{\pi}|\boldsymbol{x})$ the posterior distribution of the theory and proofs conditioned on the logical forms. The reasoning module adds the new logical form x^* to $\boldsymbol{x}$, initializes a simple proof of x^* (in the figure below, this is the trivial proof $x^* \vdash x^*$), and adds this initial proof to $\boldsymbol{\pi}$. Then, it resamples fragments of the proofs $\boldsymbol{\pi}$ until the Markov chain has mixed (the samples have approached the true posterior). Finally, when the allotted time is up, the system is evaluated, for example on a question-answering task.

The figure contains natural deduction proofs. Section A provides a brief overview of natural deduction.

3.1 Preliminary results: Active data collection

As preliminary work, we show that it is possible to algorithmically find a sequence of documents and sentences to read such that the reading competence and knowledge in the language and reasoning modules improve over time, i.e. *active data collection*. We show this by example, using the question-answering task in figure 3.1. Suppose in this example, we initialize the reasoning module with a few built-in types: `state`, `country`, `city`, `river`, `lake`, `ocean`, `borders`, `area`, `population`, `length`. We also initialize the language module by providing a handful of nouns and verbs that refer to the built-in types: ("state", `state`), ("borders", `border`), etc. The notion of numbers are also built-in, so that the language module can understand sentences like "The population of Pennsylvania is 12,702,379" or "There are 50 states in the United States," given that it has learned to recognize "Pennsylvania" and "United States." This also enables specifying seed training examples for words like "largest" or "longest," paired with logical forms that take an argmax over states or rivers. This kind of supervision is quite domain general, since superlative and counting expressions are relatively frequent in natural language. In addition, we add seed training examples containing English words that play largely grammatical roles, such as interrogative words ("which", "where", etc), prepositions ("near", "to", etc), linking verbs, etc.

With this initialization, we simulate the behavior of the proposed executive module (see section 4.3): it directs the semantic parser to attempt to parse the evaluation questions. Since our parser can return partial parses, we can inspect the returned derivation tree and find the noun phrases or verbs that the parser did not recognize. Given that the first question is "Which states border New Jersey?", the parser returns that "New Jersey" is unrecognized, and the executive module searches for documents that contain the definition of "New Jersey." The Simple English Wikipedia article contains a definition in the first article: "New Jersey is one of the 50 states of the United States of America." The noun phrase "United States of America" is also unrecognized in this definition. The executive module can retrieve the Simple English Wikipedia article for the US. The first sentence of this article is "The United States of America (USA), often called the United States (U.S.) or America, is the second largest country in North America." Given the seed training data described above, the language module should be able to fully understand this sentence, and the reasoning module adds the parsed definition as a belief. From the same article, the fact that the US has 50 states may be extracted, as well as a link to the article containing a list of the states. However, for the question-answering task, it is not necessary to parse the rest of this article. The

Initialization:

1. The reasoning module initializes the theory K to an initial state.
2. The language module is trained with a set of seed examples: {("state", `state`), ("country", `country`), ("in", `contains`)}.
3. The executive module is given a time limit, say 10 days, and a task, such as a set of questions: {"What is the smallest state bordering the Mississippi?", "Which states border New Jersey?"}.

Iteration 1: Executive module selects to read sentence "Pennsylvania is a state."

1. Language module parses sentence into $\texttt{state}(?_1)$, which is a statement that the unrecognized entity $?_1$ is a state.
2. Executive module inspects the derivation tree from language module and: (a) directs reasoning module to create new entity `pa`, (b) produces new training example for language module: ("Pennsylvania is a state", $\texttt{state}(\texttt{pa})$), and (c) adds $\texttt{state}(\texttt{pa})$ as an observation to the reasoning module (note here we chose the name `pa` conveniently for demonstration).
3. Reasoning module modifies the theory K by adding the logical form $\texttt{state}(\texttt{pa})$ and the (trivial) proof $\texttt{state}(\texttt{pa}) \vdash \texttt{state}(\texttt{pa})$.

Iteration 2: Executive module selects to read sentence "New Jersey and Pennsylvania are states in the US."

1. Language module parses sentence into $\texttt{state}(?_1) \wedge \texttt{state}(\texttt{pa}) \wedge \texttt{contains}(?_2, ?_1) \wedge \texttt{contains}(?_2, \texttt{pa})$.
2. Executive module inspects the derivation tree from language module and directs system to read definition of "US."
3. Language module re-parses sentence into $\texttt{state}(?_1) \wedge \texttt{state}(\texttt{pa}) \wedge \texttt{contains}(\texttt{us}, ?_1) \wedge \texttt{contains}(\texttt{us}, \texttt{pa})$.
4. Executive module inspects the derivation tree and: (a) directs reasoning module to create new entity `nj`, (b) adds $\texttt{state}(\texttt{nj}) \wedge \texttt{state}(\texttt{pa}) \wedge \texttt{contains}(\texttt{us}, \texttt{nj}) \wedge \texttt{contains}(\texttt{us}, \texttt{pa})$ as an observation to the reasoning module, and (c) produces new example for language module: ("New Jersey and Pennsylvania are states in the US.", $\texttt{state}(\texttt{nj}) \wedge \texttt{state}(\texttt{pa}) \wedge \texttt{contains}(\texttt{us}, \texttt{nj}) \wedge \texttt{contains}(\texttt{us}, \texttt{pa})$).
5. Reasoning module modifies the theory K by adding the new logical form and the proof:

$$\frac{}{\texttt{state}(\texttt{nj}) \wedge \texttt{state}(\texttt{pa}) \wedge \texttt{contains}(\texttt{us}, \texttt{nj}) \wedge \texttt{contains}(\texttt{us}, \texttt{pa})}\,\text{Ax}$$

which, say after a number of MCMC iterations, yields the proof:

$$\dfrac{\dfrac{\dfrac{}{\texttt{state}(\texttt{nj})}\,\text{Ax} \quad \dfrac{}{\texttt{state}(\texttt{pa})}\,\text{Ax}}{\texttt{state}(\texttt{nj}) \wedge \texttt{state}(\texttt{pa})}\,\wedge\text{I} \quad \dfrac{\dfrac{}{\texttt{contains}(\texttt{us}, \texttt{nj})}\,\text{Ax} \quad \dfrac{}{\texttt{contains}(\texttt{us}, \texttt{pa})}\,\text{Ax}}{\texttt{contains}(\texttt{us}, \texttt{nj}) \wedge \texttt{contains}(\texttt{us}, \texttt{pa})}\,\wedge\text{I}}{\texttt{state}(\texttt{nj}) \wedge \texttt{state}(\texttt{pa}) \wedge \texttt{contains}(\texttt{us}, \texttt{nj}) \wedge \texttt{contains}(\texttt{us}, \texttt{pa})}\,\wedge\text{I}$$

1. Language module parses "The United States (US) is a country" into $\texttt{country}(?_1)$.
2. Executive module inspects the derivation tree and: (a) directs reasoning module to create new entity `us`, (b) adds $\texttt{country}(\texttt{us})$ as an observation to the reasoning module, and (c) produces new training example for language module: ("The United States (US) is a country", $\texttt{country}(\texttt{us})$).
3. Reasoning module modifies the theory K by adding the logical form $\texttt{country}(\texttt{us})$ and the (trivial) proof $\texttt{country}(\texttt{us}) \vdash \texttt{country}(\texttt{us})$.

Note: the proof trees use *natural deduction* as the deductive system, where the premises are at the top and the conclusions at the bottom. The rule Ax indicates an axiom: i.e. the logical form comes from the theory K. Thus, after MCMC, the reasoning module has 3 new logical forms in the theory K: $\texttt{state}(\texttt{nj})$, $\texttt{contains}(\texttt{us}, \texttt{nj})$, $\texttt{contains}(\texttt{us}, \texttt{pa})$.

...after 10 days: The system is evaluated on the question-answering task.

1. The language module parses the question
   "Which states border New Jersey?
   (a) New York and Pennsylvania, (b) Delaware and Maryland, or (c) New York, Pennsylvania, and Delaware"
   into $\lambda x(\texttt{state}(x) \wedge \texttt{border}(x, \texttt{nj}))$ with options: (a) $\{\texttt{ny}, \texttt{pa}\}$, (b) $\{\texttt{de}, \texttt{ma}\}$, or (c) $\{\texttt{ny}, \texttt{pa}, \texttt{de}\}$.
2. The reasoning module uses MCMC to produce s samples of K: $K_1, \ldots, K_s$. For each sample $i = 1, \ldots, s$, the reasoning module computes the denotation of the logical form with respect to the theory K_i. Suppose the most frequent denotation is the set $\{\texttt{ny}, \texttt{pa}, \texttt{de}\}$. Since this matches with option (c), we output the response (c).
3. Repeat for every question in the evaluation set...

FIGURE 3.1: An example of the end-to-end execution of our method.

executive module then goes back to the New Jersey article, and attempts to re-parse the first sentence. This time, it succeeds and adds New Jersey as an instance of `state` to the theory.

Now every token in the original question is recognized, and may be parsed into logical form. The executive module then can attempt to compute the denotation of the logical form with respect to the theory, and deduce that the theory does not contain an exclusive list of the states bordering New Jersey. To this end, it directs the language module to parse additional sentences from the New Jersey article. The third sentence is "It is a small state, shaped like a letter [S], and bordered on the west by Pennsylvania and Delaware across the Delaware River, on the north by New York, on the northeast by the Hudson River and New York City, on the east and southeast by the Atlantic Ocean, and on the southwest by Delaware Bay." Since the tokens "Pennsylvania", "Delaware", "Delaware River", etc are unrecognized, the executive module finds the relevant articles on Simple English Wikipedia: Pennsylvania, Delaware, Delaware River, etc. The executive module repeats this process, adding definitions for each of the bordering states and rivers, until it either is able to answer the evaluation question, or it fails after reaching a time limit or the maximum recursion depth. Then, it moves on to the next evaluation question, continuing until the time limit. This simple proof of concept shows that with the right seed training data, it is possible for an algorithm to find a sequence of articles and sentences that would improve the competence of the

language and reasoning modules.

3.2 Preliminary results: Language module

We have developed a semantic parsing algorithm that can incorporate background knowledge during parsing. In contrast to most semantic parsers, which are built on discriminative models, our model is fully generative: To generate a sentence, the logical form is first drawn from a prior. A grammar then recursively constructs a derivation tree top-down, probabilistically selecting production rules from distributions that depend on the logical form. The semantic prior distribution provides a straightforward way to incorporate background knowledge, such as information about the types of entities and predicates, or the context of the utterance. For example, logical forms that are consistent with the background knowledge can be assigned greater prior probability, which enables the parser to disambiguate sentences with multiple interpretations. In addition, our parser can return *partial* parses of sentences, which is useful for sentences that contain a small number of unseen words, such as definitions.

S → N:`select_arg1` VP:`delete_arg1`
VP → V:`identity` N:`select_arg2`
VP → V:`identity`
N → "New Jersey"
N → "NJ"
N → "Pennsylvania"
V → "borders"
V → "has"

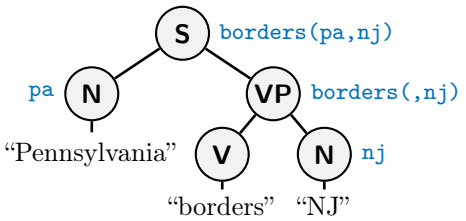


FIGURE 3.2: **(top)** Example of a grammar in our framework. This grammar operates on logical forms of the form *predicate(first argument, second argument)*. The semantic function `select_arg1` returns the first argument of the logical form. Likewise, the function `select_arg2` returns the second argument. The function `delete_arg1` removes the first argument, and `identity` returns the logical form with no change. In our use of the framework, the interior production rules (the first three listed) are examples of rules that we specify, whereas the terminal rules and the posterior probabilities of *all* rules are learned via grammar induction. We also use a richer semantic formalism than in this example. The subsection on **selecting production rules** in section 3.2.1 provides more detail. **(bottom)** Example of a derivation tree under the grammar given in Figure 3.2. The logical form corresponding to every node is shown in blue beside the respective node. The logical form for V is `borders(,nj)` and is omitted to reduce clutter.

3.2.1 Semantic parsing model

A grammar in our formalism operates over a set of nonterminals $\mathcal{N}$ and a set of terminal symbols $\mathcal{W}$. It can be understood as an extension of a context-free grammar (CFG) [8] where the generative process for the syntax is dependent on a logical form, thereby coupling syntax with semantics. In the top-down generative process of a derivation tree, a logical form guides the selection of production rules. Production rules in our grammar have the form $A \to B_1{:}f_1 \dots B_k{:}f_k$ where $A \in \mathcal{N}$ is a nonterminal, $B_i \in \mathcal{N} \cup \mathcal{W}$ are right-hand side symbols, and f_i are *semantic transformation functions*. These functions can encode how to "decompose" this logical form when recursively generating the subtrees rooted at each B_i. Thus, they enable semantic compositionality. An example of a grammar in this framework and a derivation tree is shown in Figure 3.2. Let $\boldsymbol{\mathcal{R}}$ be the set of production rules in the grammar and $\boldsymbol{\mathcal{R}}_A$ be the set of production rules with left-hand nonterminal symbol A.

Generative process: A *parse tree* (or *derivation*) in this formalism is a tree where every interior node is labeled with a nonterminal symbol, every leaf is labeled with a terminal, and the root node is labeled with the root nonterminal S. Moreover, every node in the tree is associated with a logical form: let $x^{\boldsymbol{n}}$ be the logical form assigned to the tree node $\boldsymbol{n}$, and $x^{\mathbf{0}} = x$ for the root node $\mathbf{0}$.

The generative process to build a parse tree begins with the root nonterminal S and a logical form x. We *expand* S by randomly drawing a production rule from $\boldsymbol{\mathcal{R}}_S$, *conditioned* on the logical form x. This provides the first level of child nodes in the derivation tree. So if, for example, the rule $S \to B_1{:}f_1 \dots B_k{:}f_k$ were drawn, the root node would have k child nodes, $\boldsymbol{n}_1, \dots, \boldsymbol{n}_k$, respectively labeled $B_1, \dots, B_k$. The logical form associated with each node is determined by the semantic transformation function: $x^{\boldsymbol{n}_i} = f_i(x)$. These functions describe the relationship between the logical form at a child node and that of its parent node. This process repeats recursively with every right-hand side nonterminal symbol, until there are no unexpanded nonterminal nodes. The sentence is obtained by taking the *yield* of the terminals in the tree (a concatenation).

The semantic transformation functions are specific to the semantic formalism and may be defined as appropriate to the application. In our parsing application, we define a domain-independent set of transformation functions (e.g., one function selects the left n conjuncts in a conjunction, another selects the n^{th} argument of a predicate instance, etc).

Selecting production rules: In the above description, we did not specify the distribution from which rules are selected from $\boldsymbol{\mathcal{R}}_A$. There are many modeling options available when specifying this distribution. In our approach, we choose a hierarchical Dirichlet process (HDP) prior [34]. Every nonterminal in our grammar $A \in \mathcal{N}$ will be associated with an HDP hierarchy. For each nonterminal, we specify a sequence of *semantic feature functions*, $\{g_1, \ldots, g_m\}$, each of which return a discrete feature (such as an integer) of an input logical form x. We use this sequence of feature functions to define the hierarchy of the HDP: starting with the root node, we add a child node for every possible value of the first feature function g_1. For each of these child nodes, we add a grandchild node for every possible value of the second feature function g_2, and so forth. The result is a complete tree of depth m. Each node $\boldsymbol{n}$ in this tree is assigned a distribution $G^{\boldsymbol{n}}$:

$$G^{\mathbf{0}} \sim \mathrm{DP}(\alpha^{\mathbf{0}}, H), \qquad G^{\boldsymbol{n}} \sim \mathrm{DP}(\alpha^{\boldsymbol{n}}, G^{\pi(\boldsymbol{n})}), \tag{3.1}$$

where $\mathbf{0}$ is the root node, $\pi(\boldsymbol{n})$ is the parent of $\boldsymbol{n}$, α are a set of concentration parameters, and H is a base distribution over $\boldsymbol{\mathcal{R}}_A$. This base distribution is independent of the logical form x.

To select a rule in the generative process, given the logical form x, we can compute its feature values $(g_1(x), \ldots, g_m(x))$ which specify a unique path in the HDP hierarchy to a leaf node G^x. We then draw the production rule from G^x. The specified set of production rules and semantic features are included with the code package. The specified rules and features do not change across our experiments.

Take, for example, the derivation tree in Figure 3.2. In the generative process where the node VP is expanded, the production rule is drawn from the HDP associated with the nonterminal VP. Suppose the HDP was constructed using a sequence of two semantic features: (`predicate`, `arg2`). In the example, the feature functions are evaluated with the logical form `borders(,nj)` and they return the sequence (`borders`, `nj`). This sequence uniquely identifies a path in the HDP hierarchy from the root node $\mathbf{0}$ to a leaf node $\boldsymbol{n}$. The production rule VP $\rightarrow$ V N is drawn from this leaf node $G^{\boldsymbol{n}}$, and the generative process continues recursively.

In our implementation, we divide the set of nonterminals $\mathcal{N}$ into two groups: (1) the set of "interior" nonterminals, and (2) preterminals. The production rules of preterminals are restricted such that the right-hand side contains only terminal symbols. The rules of interior nonterminals are restricted such that only nonterminal symbols appear on the right side.

1. For **preterminals**, we set H to be a distribution over sequences of terminal symbols as follows: we generate each token in the sequence i.i.d. from a uniform distribution over a finite set of terminals and a special *stop* symbol with probability ϕ_A. Once the stop symbol is drawn, we have finished generating the rule. Note that we do not specify a set of domain-specific terminal symbols in defining this distribution.
2. For **interior nonterminals**, we specify H as a discrete distribution over a domain-independent set of production rules. This requires specifying a set of nonterminal symbols, such as S, NP, VP, etc. Since these production rules contain semantic transformation functions, they are specific to the semantic formalism.

We emphasize that only the prior is specified here, and we will use grammar induction to infer the posterior. In principle, a more relaxed choice of H may enable grammar induction without pre-specified production rules, and therefore without dependence on a particular semantic formalism or natural language, if an efficient inference algorithm can be developed in such cases.

3.2.2 Training

We describe grammar induction independently of the choice of rule distribution. Let $\boldsymbol{\theta}$ be the random variables in the grammar: in the case of the HDP prior, $\boldsymbol{\theta}$ is the set of all distributions $G^{\boldsymbol{n}}$ at every node in the hierarchies. Given a set of sentences $\boldsymbol{y} \triangleq \{y_1, \ldots, y_n\}$ and corresponding logical forms $\boldsymbol{x} \triangleq \{x_1, \ldots, x_n\}$, we wish to compute the posterior $p(\boldsymbol{t}, \boldsymbol{\theta} | \boldsymbol{x}, \boldsymbol{y})$ over the unobserved variables: the grammar $\boldsymbol{\theta}$ and the latent derivations/parse trees $\boldsymbol{t} \triangleq \{t_1, \ldots, t_n\}$. This is intractable to compute exactly, and so we resort to Markov chain Monte Carlo (MCMC) [14, 27]. To perform blocked Gibbs sampling, we pick initial values for $\boldsymbol{t}$ and $\boldsymbol{\theta}$ and repeat the following:

1. For $i = 1, \ldots, n$, sample $t_i | \boldsymbol{\theta}, x_i, y_i$.
2. Sample $\boldsymbol{\theta} | \boldsymbol{t}$.

However, since the sampling of each tree t depends on $\boldsymbol{\theta}$, and we need to resample all n parse trees before sampling $\boldsymbol{\theta}$, this Markov chain can be slow to mix. Thus, we employ collapsed Gibbs sampling by integrating out $\boldsymbol{\theta}$. In this algorithm, we repeatedly sample from $t_i | \boldsymbol{t}_{-i}, x_i, y_i$ where $\boldsymbol{t}_{-i} = \boldsymbol{t} \setminus \{t_i\}$.

$$p(t_i | \boldsymbol{t}_{-i}, x_i, y_i) = \mathbb{1}\{\mathrm{yield}(t_i) = y_i\} \prod_{A \in \mathcal{N}} p\left(\bigcap_{\substack{\{\boldsymbol{n} \in t_i \,:\, \boldsymbol{n} \\ \text{has label } A\}}} r^{\boldsymbol{n}} \;\middle|\; \boldsymbol{t}_{-i}, x_i \right), \tag{3.2}$$

where the intersection is taken over tree nodes $\boldsymbol{n} \in t_i$ labeled with the nonterminal A, $r^{\boldsymbol{n}}$ is the production rule at node $\boldsymbol{n}$, $\mathrm{yield}(t_i)$ concatenates the terminals at the leaves of the tree t_i, and $\mathbb{1}\{\cdot\}$ is 1 if the condition is true and zero otherwise. With $\boldsymbol{\theta}$ integrated out, the probability does not necessarily factorize over

rules. In the case of the HDP prior, selecting a rule will increase the probability that the same rule is selected again (due to the "rich get richer" effect observed in the Chinese restaurant process). We instead use a Metropolis-Hastings step to sample t_i, where the proposal distribution is given by the fully factorized form:

$$p(t_i^*|\boldsymbol{t}_{-i}, x_i, y_i) = \mathbb{1}\{\text{yield}(t_i^*) = y_i\} \prod_{\boldsymbol{n}\in t_i^*} p\left(r^{\boldsymbol{n}} \mid \boldsymbol{t}_{-i}, x_i^{\boldsymbol{n}}\right). \tag{3.3}$$

After sampling t_i^*, we choose to accept the new sample with probability

$$\frac{\prod_{\boldsymbol{n}\in t_i} p(r^{\boldsymbol{n}}|x^{\boldsymbol{n}}, \boldsymbol{t}_{-i})}{p\left(\bigcap_{\boldsymbol{n}\in t_i} r^{\boldsymbol{n}}|x, \boldsymbol{t}_{-i}\right)} \frac{p\left(\bigcap_{\boldsymbol{n}\in t_i^*} r^{\boldsymbol{n}}|x, \boldsymbol{t}_{-i}\right)}{\prod_{\boldsymbol{n}\in t_i^*} p(r^{\boldsymbol{n}}|x^{\boldsymbol{n}}, \boldsymbol{t}_{-i})},$$

where t_i, here, is the old sample, and t_i^* is the newly proposed sample. In practice, this acceptance probability is very high. This approach is very similar in structure to that in [17, 2, 9].

If an application requires posterior samples of the grammar variables $\boldsymbol{\theta}$, we can obtain them by drawing from $\boldsymbol{\theta}|\boldsymbol{t}$ after the collapsed Gibbs sampler has mixed. Note that this algorithm requries no further supervision beyond the utterances $\boldsymbol{y}$ and logical forms $\boldsymbol{x}$. However, it is able to exploit additional information such as supervised derivations/parse trees. For example, a lexicon can be provided where each entry is a terminal symbol y_i with a corresponding logical form label x_i. We evaluate our method with and without such a lexicon. To sample from equation (3.3), we use inside-outside sampling [13, 17], a dynamic programming approach. For details, refer to [29].

3.2.3 Parsing

For a new sentence y_*, we aim to find the logical form x_* and derivation t_* that maximizes

$$p(x_*, t_*|y_*, \boldsymbol{\theta}) \propto p(x_*)p(y_*|t_*)p(t_*|x_*, \boldsymbol{\theta}) = \mathbb{1}\{\text{yield}(t_*) = y_*\}p(x_*)\prod_{\boldsymbol{n}\in t_*} p(r^{\boldsymbol{n}}|x_*^{\boldsymbol{n}}, \boldsymbol{\theta}). \tag{3.4}$$

Here, $\boldsymbol{\theta}$ is a point estimate of the grammar, which may be obtained from a single sample, or from a Monte Carlo average over a finite set of samples.

Naively computing the above probability for every possible x_* and t_* is intractable. Instead, we use a branch-and-bound approach, dividing the problem into subproblems of finding logical forms and derivations for phrases within the sentence y_*. We compute an upper bound on the posterior probability for each subproblem, and we can ignore a very large number of subproblems whose upper bound is too high. Figure 3.4 shows the search tree for the branch-and-bound algorithm. By ignoring search states with an upper bound smaller than that of the highest-scoring state in the search queue, the parser can ignore a large number of improbable logical forms and derivations. Thus, with a good upper bound, the parser can run in sublinear time with respect to the size of the knowledge base. In our approach, in order to compute a tighter upper bound, we first perform a syntactic parse of the input sentence. That is, for every nonterminal A and every sentence span (i, j), we compute

$$I_{(A,i,j)} \triangleq \max_{A\to B_1\ldots B_K} \left(\max_{x'} \log p(A \to B_1, \ldots, B_K|x', \boldsymbol{\theta}) + \max_{l_2<\ldots<l_K} \sum_{k=1}^{K} I_{(B_k,l_k,l_{k+1})} \right),$$

where $l_1 = i$, $l_{K+1} = j$. Note that the left term is a maximum over all logical forms x', and so this upper bound only considers syntactic information. The right term can be maximized using dynamic programming in $\mathcal{O}(K^2)$. As such, classical syntactic parsing algorithms can be applied to compute I for every chart cell in $\mathcal{O}(n^3)$. $I_{(A,i,j)}$ can then be used to be compute a tighter upper bound on the log posterior in the branch-and-bound algorithm. The parser is guaranteed to find derivations in order of non-increasing posterior probability, and so it can be used to compute the top-k Viterbi parses. Refer to [29] for details.

Method	Additional Supervision	GeoQuery P	GeoQuery R	GeoQuery F1	Jobs P	Jobs R	Jobs F1
WASP [37]	A,B	87.2	74.8	80.5			
λ-WASP [38]	A,B,F	92.0	86.6	89.2			
Extended GHKM [21]	B,F	93.0	87.6	90.2			
Zettlemoyer and Collins [39]	C,E,F	**96.3**	79.3	87.0	97.3	79.3	87.4
Zettlemoyer and Collins [40]	C,E,F	91.6	86.1	88.8			
UBL [18]	E	94.1	85.0	89.3			
FUBL [19]	E	88.6	88.6	88.6			
TISP [41]	E,F	92.9	**88.9**	**90.9**	85.0	**85.0**	85.0
GSG − lexicon − type-checking	D	86.9	75.7	80.9	89.5	67.1	76.7
GSG + lexicon − type-checking	D,E	88.4	81.8	85.0	91.4	75.7	82.8
GSG − lexicon + type-checking	D,F	89.3	77.9	83.2	93.2	69.3	79.5
GSG + lexicon + type-checking	D,E,F	90.7	83.9	87.2	**97.4**	81.4	**88.7**

Legend for sources of additional supervision are:
A. Training set containing 792 examples,
B. Domain-specific set of initial synchronous CFG rules,
C. Domain-independent set of lexical templates,
D. Domain-independent set of interior production rules,
E. Domain-specific initial lexicon,
F. Type-checking and type specification for entities.

FIGURE 3.3: The methods in the top part of the table were evaluated using 10-fold cross validation, whereas those in the bottom part were evaluated with an independent test set.

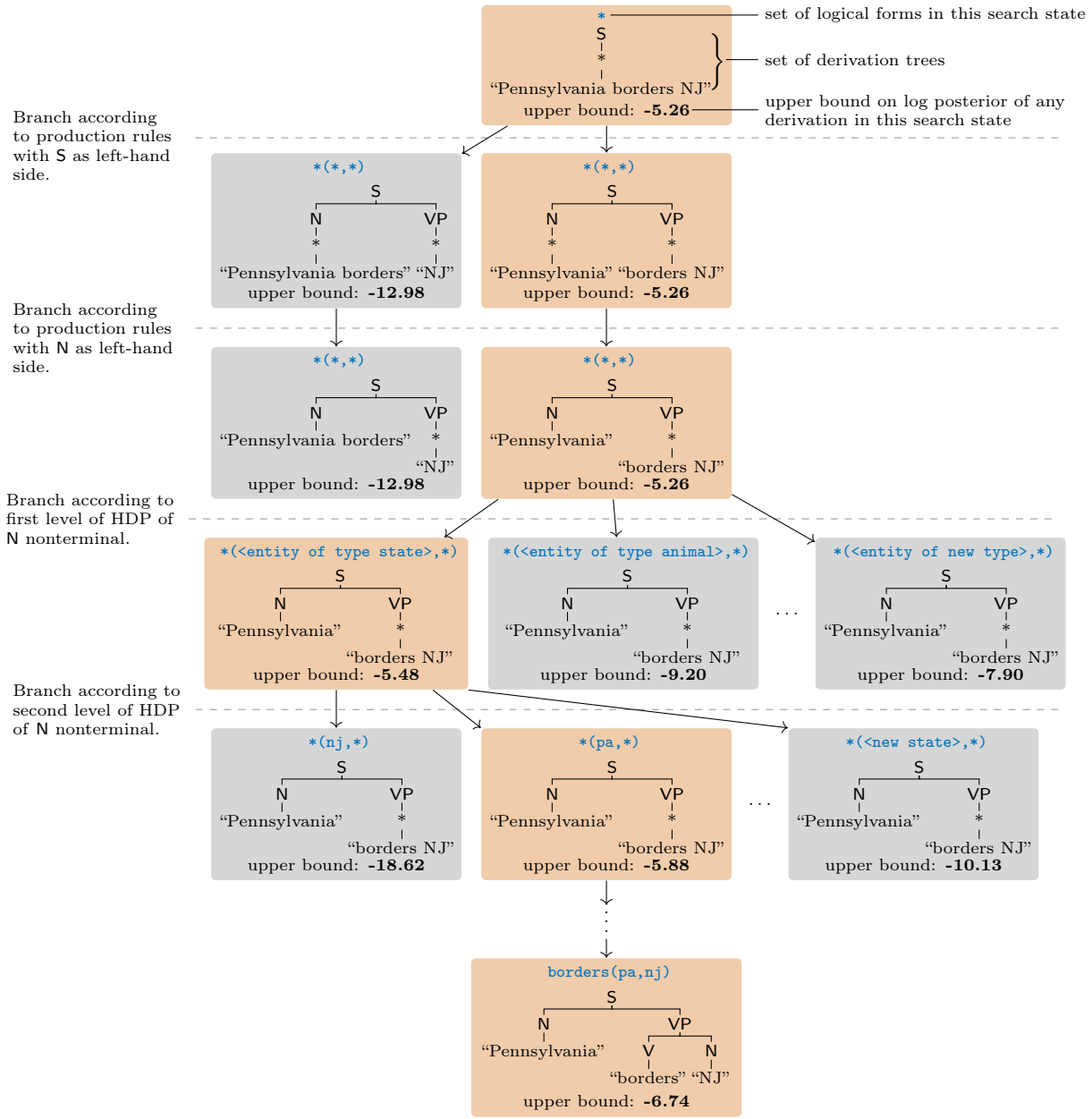


FIGURE 3.4: The search tree of the branch-and-bound algorithm during parsing. In this diagram, each square is a search state, representing a set of logical forms and derivations. * denotes the set of all possible logical forms, whereas * denotes the set of all possible derivation trees. The gray-colored search states are unvisited by the parser, since their upper bounds on the log posterior are smaller than that of the completed parse at the bottom of the diagram (-6.74), thus allowing the parser to ignore a very large number of improbable logical forms and derivations. In this example, we use the grammar from figure 3.2 and assume that the HDP for the N nonterminal has a depth of 2, where the first layer is constructed according to the *semantic type* of the entity, and the second layer is constructed according to the entity itself. This enables our parser to ignore improbable *types* of entities which can further reduce the number of states we need to explore.

We evaluate our parser on the GEOQUERY and JOBS datasets. GEOQUERY is a set of 880 questions about U.S. geography and JOBS is a set of 640 queries to a software engineering job search engine. In both datasets, every sentence is labeled with a Datalog logical form. Results are shown in figure 3.3 where our algorithm is called GSG. Our implementation is available for reference at https://github.com/asaparov/parser.

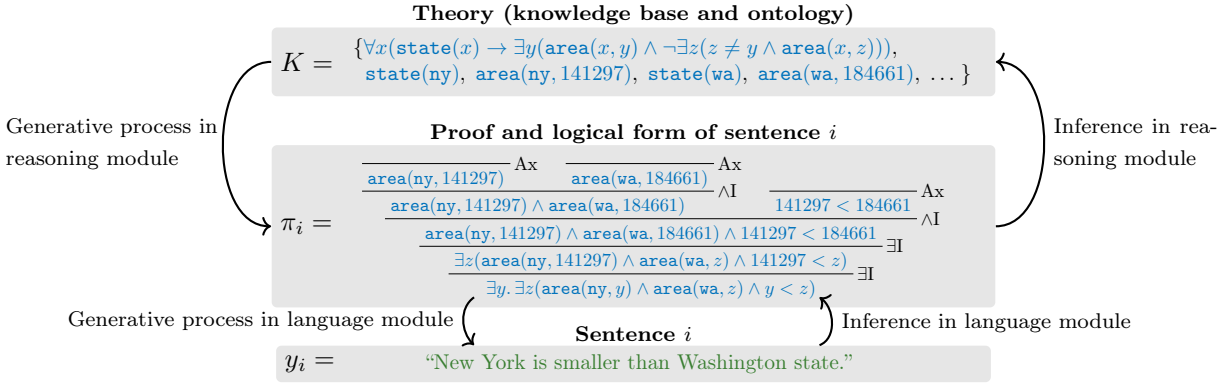


FIGURE 4.1: Schematic of the generative process and inference in our system, with an example of a theory containing facts about U.S. geography, generating a proof of a logical form which, in turn, is used to generate the sentence "New York is smaller than Washington state." During inference, given the sentence, the language module outputs the logical form. The reasoning module must then infer the proof π_i and the theory K.

4. PROPOSED WORK

4.1 Reasoning module

In order to use statistical methods to learn K and π, we need to specify the probability $p(K, \boldsymbol{\pi}) = p(K) \prod_i p(\pi_i|K)$. We first discuss the term for proofs $p(\pi_i|K)$ and present the inference algorithm. After doing so, we will discuss distributions over the theory $p(K)$.

Recall that each proof π_i is a sequence of logical forms $(\phi_1, \ldots, \phi_k)$, where each logical form ϕ_i is annotated with a deduction rule d_i. The rule d_i was applied to a set a premises or axioms to obtain ϕ_i as its conclusion. Thus, we can characterize proofs as directed acyclic graphs. Note that in general, a DAG may have multiple valid topological orderings of its vertices. Similarly, the deduction steps in a proof may be ordered in many different ways. *Exchangeability* is an important property that is required for efficient statistical inference. We say the distribution $p(\pi)$ is *exchangeable* if $p(\pi) = p(\sigma(\pi))$ for every permutation of the deduction steps σ that preserves the topological ordering of π. One easy way to obtain an exchangeable distribution is to impose a *canonical ordering* of the deduction steps: Establish an arbitrary but fixed ordering on the types of deduction rules (e.g. $\text{Ax} \prec \wedge\text{I} \prec \wedge\text{E}_L \prec \ldots$), and simply topologically sort the deduction steps in the proof, breaking ties according to the fixed ordering of deduction rules. Thus, all valid topological reorderings of a given proof will be sorted into a single ordering. To mitigate potential identifiability issues, we impose further restrictions on proofs. A proof π is called *canonical* if: (1) π is canonically-ordered, (2) for every distinct pair of logical forms $\phi_j, \phi_{j'} \in \pi$ where $j \neq j'$ that share the same hypotheses, $\phi_j \neq \phi_{j'}$, and (3) for every logical form except the last $\phi_j \in \pi$ where $j < |\pi|$, ϕ_j is used as a premise for some rule in π.

Generative process for proofs: In our first implementation attempt, we will assume a fairly simple distribution for $p(\pi_i|K)$. We specify this distribution by providing a generative process for the proof $\pi_i = (\pi_{i,1}, \ldots, \pi_{i,k})$, given K. First, we generate a proof $\bar{\pi}_i$ that may not be canonical. So for $j = 1, 2, \ldots$:

1. Select a deduction rule $d_j \in \mathcal{D}$. If d_j is the Ax rule, then sample a logical form uniformly from K. If d_j requires antecedents, then we select each antecedent uniformly at random from $\bar{\pi}_{i,1}, \ldots, \bar{\pi}_{i,j-1}$. Then set $\bar{\pi}_{i,j}$ to be the conclusion of the application of d_j.
2. If the current proof $\bar{\pi}_{i,1}, \ldots, \bar{\pi}_{i,j}$ has a single conclusion, i.e. every logical form in $\bar{\pi}_{i,1}, \ldots, \bar{\pi}_{i,j-1}$ is used as an antecedent to some deduction in the proof (or equivalently, the DAG has exactly one sink vertex), then with probability α, we stop generating the proof.

At the end of this process, if the proof $\bar{\pi}_i = (\bar{\pi}_{i,1}, \ldots, \bar{\pi}_{i,k})$ is not canonical, then the process fails and we retry by generating a new proof. Otherwise, we set $\pi_i = \bar{\pi}_i$. Thus, the probability of generating a proof $p(\pi_i|K)$ is given by:

$$p(\pi_i|K) = \frac{\mathbb{1}\{\bar{\pi}_i \text{ canonical}\}}{p(\bar{\pi}_i \text{ canonical})} p(\bar{\pi}_i|K) = \alpha \frac{\mathbb{1}\{\bar{\pi}_i \text{ canonical}\}}{p(\bar{\pi}_i \text{ canonical})} \prod_{j=1}^{|\pi_i|} p(\bar{\pi}_{i,j}|\bar{\pi}_{i,1}, \ldots, \bar{\pi}_{i,j-1}, K) \cdot (1-\alpha)^{\mathbb{1}\{\bar{\pi}_{i,1},\ldots,\bar{\pi}_{i,j-1} \text{ has a single conclusion}\}}.$$

The canonical ordering ensures that $p(\pi_i|K)$ is exchangeable, even if $p(\bar{\pi}_i|K)$ is not. Note, however, this expression is difficult to compute due to the term $p(\bar{\pi}_i \text{ canonical})$.

Inference: We use Metropolis-Hastings to perform inference in this model, since it allows us to avoid computing the expensive normalization term. Given a set of logical forms $x_1, \ldots, x_n$ (provided by the language module), we want to infer the posterior distribution $p(K, \boldsymbol{\pi}|\boldsymbol{x})$. To do so, we begin with initial values for K and $\boldsymbol{\pi}$. We must be careful to make sure that each π_i is a valid proof of x_i, and all the axioms in π_i exist in K. If $p(K)$ has support over all logical forms in $\boldsymbol{x}$, i.e. $p(x_i \in K) > 0$ for all x_i, then a trivial way to initialize K and $\boldsymbol{\pi}$ is to simply add the observed logical forms $\boldsymbol{x}$ to K. Then, each

proof π_i can be initialized as a trival proof of x_i, containing a single Ax rule and nothing else: $x_i \vdash x_i$. However, most interesting priors for the theory $p(K)$ will not have support over all logical forms, and we will need to devise methods to initialize K and $\boldsymbol{\pi}$ specially for those priors.

Once initialized, we select a single deduction step $\pi_{i,j}$ within a proof to resample (either randomly or in order). Next, we select a *local transformation* to apply to the proof. We do so by sampling from a set of available transformations. This transformation will alter the proof, affecting only the steps near and including $\pi_{i,j}$, and *without changing the conclusion* of $\pi_{i,j}$. These transformations preserve the conclusion but may alter the antecedents (and by extension, the elements of K), since during inference time, the conclusion of each proof x_i is known and fixed, but the antecedents ultimately come from the theory K, which is unknown and may vary. Our proposed set of transformations are listed in figure B.1. Let π_i^* be the transformed (and canonicalized) proof and K^* is the altered theory as a result of the transformation. Then, with probability $Q(\boldsymbol{\pi}, K|\boldsymbol{\pi}^*, K^*)p(\boldsymbol{\pi}^*, K^*)/(Q(\boldsymbol{\pi}^*, K^*|\boldsymbol{\pi}, K)p(\boldsymbol{\pi}, K))$ we accept the transformed proofs and theory as the new sample. Otherwise, we reject the proposal. $Q(\boldsymbol{\pi}^*, K^*|\boldsymbol{\pi}, K)$ is the probability of proposing the transformation $\boldsymbol{\pi}^*, K^*$ given that the previous sample is $\boldsymbol{\pi}, K$. We repeat this sampling process until the Markov chain mixes and converges to the posterior.

However, in order for the Markov chain to converge to the posterior, it must be *irreducible*. Let $\boldsymbol{\pi}^{(0)}, K^{(0)}$ be the initial values of $\boldsymbol{\pi}$ and K in the Markov chain. We say the Markov chain is irreducible if for any $\boldsymbol{\pi}, K$ such that $p(\boldsymbol{\pi}, K|\boldsymbol{x}) > 0$, there exists a sequence of proof transformations with non-zero probability that transform $\boldsymbol{\pi}^{(0)}, K^{(0)}$ into $\boldsymbol{\pi}, K$. Phrased differently, every part of the posterior must be reachable from the initial state. Thus, to make the Markov chain irreducible, we need to ensure that the set of available transformations allow the algorithm to explore a sufficiently large portion of the space of all proofs $\boldsymbol{\pi}$ and theories K. If we choose a prior for K such that $p(x \in K) > 0$ for any logical form x, then the transformations in figure B.1 yield an irreducible Markov chain. This is true since the transformations can be used to "construct" any canonical proof of x, starting from the trivial proof $x \vdash x$ and iteratively applying the transformations at the leaf nodes (Ax rules) to grow the proof tree. However, in the below section, we present more interesting and structured priors for K in which $p(x \in K) > 0$ is not always true. In such cases, additional transformations may be necessary to ensure irreducibility of the Markov chain.

The posterior $p(\boldsymbol{\pi}, K|\boldsymbol{x})$ is likely highly multi-modal, and it may be difficult to explore the space of $\boldsymbol{\pi}$ and K with a single Markov chain. So we can use multiple Markov chains to explore different modes of the posterior.

Priors on the theory: An important property for the theory K is *consistency*, i.e. a contradiction is not provable $K \nvdash \bot$. We require this property to hold in the model, but we do not necessarily need to impose this property during inference. In general, we wish to place higher probability on smaller theories, which is essentially *Occam's razor*, where we prefer simpler explanations for the phenomena we observe. Perhaps the simplest prior for K is one in which the logical forms are drawn uniformly from some predetermined set $\mathcal{X}$. This set may be the set of all logical forms with bounded depth, for example. The generative process starts by selecting the size of K from a geometric distribution. Then each logical form in K is drawn uniformly from $\mathcal{X}$. If the resulting theory is inconsistent or if the selected logical forms are not unique, then fail and restart the process. In this simple model, K has very little structure, but it may be a good place to start since implementation is straightforward.

In reality, facts about the world, e.g. in Wikipedia, are highly structured, and the prior for K should ideally prefer the same kind of structure. T should be sparse, where the vast majority of pairs of concepts are not directly connected by any logical form. This sparsity is hierarchical, where concepts within the same domain are more likely to be related than if they were in distinct domains. To incorporate these assumptions, divide T into two parts: the ontology and the knowledge base (KB). The ontology contains beliefs about types of objects and relations, whereas the KB contains beliefs about instances thereof. The types in the ontology form a hierarchy (DAG) of finite depth, with a single "object" type at the root. Every type $\mathtt{t}$ in the ontology has a small finite number of sentences of the form

$$\forall x(\mathtt{type}(x, \mathtt{t}) \wedge f(x) \rightarrow \mathtt{type}(x, \mathtt{p}_1) \wedge \ldots \wedge \mathtt{type}(x, \mathtt{p}_k) \wedge g(x)),$$

where $\mathtt{p}_1, \ldots, \mathtt{p}_k$ are the parents of $\mathtt{t}$ and $g(x)$ is an expression that qualifies all x of type $\mathtt{t}$ that satisfy the expression $f(x)$. To generate such a theory K, we first generate the hierarchy of the types in K, then draw a handful of sentences for each type of the kind described above.

To add further structure to the prior, we can constrain $g(x)$ to be non-empty, and every type $\mathtt{t}$ has a unique *definitional* statement of the form

$$\forall x(\mathtt{type}(x, \mathtt{t}) \leftrightarrow \mathtt{type}(x, \mathtt{p}_1) \wedge \ldots \wedge \mathtt{type}(x, \mathtt{p}_k) \wedge h(x)),$$

where $h(x)$ is an expression that further qualifies x. We can add the further constraint that definitions be non-circular, where $h(x)$ cannot rely on concepts whose definitions depend on $\mathtt{t}$.

Another variation of the above priors is to impose further structure on relations. For example, a relation type $\mathtt{r}$ may require at least one argument: $\forall x(\mathtt{type}(x, \mathtt{r}) \rightarrow \exists y.\, \mathtt{arg1}(x, y))$. Other relation types may require the *absence* of arguments (unary relation types, for example). The arguments of relations

may be typed as well: $\forall x.\forall y(\texttt{type}(x,\texttt{r}) \wedge \texttt{arg1}(x,y) \rightarrow \texttt{type}(y,\texttt{t}))$. These constraints are realized in the sentences of the forms described above, and thus are automatically inherited by descendant relation types.

We require a number of axioms be satisfied by all of the above priors (they are included into K by default). The transitivity of the `type` predicate is critical:

$$\forall x.\,\forall y.\,\forall z(\texttt{type}(x,y) \wedge \texttt{type}(y,z) \rightarrow \texttt{type}(y,z)).$$

In addition, all relation instances must have at most one `arg1` edge, at most one `arg2` edge, etc.

$$\neg\exists x.\,\exists y.\,\exists z(\texttt{arg1}(x,y) \wedge \texttt{arg1}(x,z) \wedge y \neq z), \text{ and similarly for } \texttt{arg2}, \text{ etc.}$$

4.2 Semantic representation

The representation of logical forms that represent the meaning of sentences is a key design problem. There is a clear tradeoff between the expressive power of a formal language and the tractability of reasoning with that language, as well as the difficulty of implementing and working with the language in code. In addition, our language model performs better if the structure of the logical forms is as close as possible to the structure of the sentences. It is for this reason that plain first-order logic or other purely logical languages are not appropriate. The structure of a sentence can be extraordinarily divergent from the structure of the equivalent first-order expression. Formalisms such as AMR [1, 4] and formalisms based on universal dependencies (UD) [36, 26] provide a very close relationship between the semantic and syntactic structure of a sentence, but they lack expressivity. For example, AMR cannot express sentences with universal quantification. Without universal quantification in the formal language, the reasoning module would be unable to generalize or perform inductive learning.

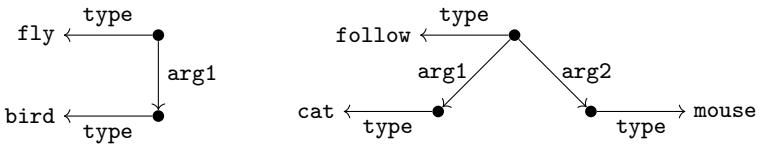
Therefore, in this section, we present a handful of formal languages to represent logical forms in our system. We start with a simple but restrictive formalism, and progressively add extensions to the formalism. We define the semantics of each formalism by providing a translation function that can convert any sentence in the formalism into a sentence in first-order logic. Our translations rely on set theory, since it provides a nice way to incorporate the more complex forms of quantification that arise in natural language. This will become apparent when we introduce universal quantification in 4.2.3 into the language, but the following is a simple example demonstrating the utility of the set-theoretic formalism: "Everyone ate a pastry. They were sweet." Here, even though "pastry" is singular, the second sentence quantifies over the set of *all eaten pastries*. If we wish for our language module to be able to read sentences like these, the logical formalism must be able to express this kind of quantification.

4.2.1 Simple logical formalism

The simplest formalism we present contains only conjunction and existential quantification. These logical

Natural language	**Logical form**
"a bird flies"	`f:type(fly) arg1(b:type(bird))`
"a cat follows a mouse"	`f:type(follow) arg1(c:type(cat)) arg2(m:type(mouse))`

forms may also be represented as graphs, as in figure 4.2. Note that each variable in the logical form corresponds to a vertex in the graph, and each predicate corresponds to an edge.



"a bird flies" "a cat follows a mouse"

FIGURE 4.2: Graph representations of the logical forms for "a bird flies" and "a cat follows a mouse."

We translate the logical form into first-order logic by converting every variable into a *set*. We define the set to contain all possible assignments of the variable to objects that will satisfy the logical form. Thus, in the first example above, the variable `b` will be translated into a set S_b such that every $b \in S_b$ is a bird that is flying:

$$\begin{aligned}\exists S_b.\,\exists S_f(&\forall b(b \in S_b \leftrightarrow \text{type}(b,\text{bird}) \wedge \exists f \in S_f(b)) \wedge \\ &\forall f.\,\forall b(f \in S_f(b) \leftrightarrow \text{type}(f,\text{fly}) \wedge \text{arg1}(f,b)) \wedge \\ &\exists b \in S_b.\,\exists f \in S_f(b)).\end{aligned}$$

Also note that S_f is a set-valued function, where $S_f(b)$ is the set of "flying events" in which b is the first argument. The sets S_b and S_f are useful, since it enables quantification over the set of all flying birds by subsequent sentences, for example. This kind of quantification would not be possible without these sets.

In the second example, the variables `f`, `c`, and `m` will each be translated into a set: F, C, and M, respectively. The set C will contain all cats that follow some mouse $m \in M$, and the set M will contain all mice that are followed by some cat $c \in C$.

$$\begin{aligned}\exists S_c.\, \exists S_m.\, \exists S_f(&\forall c(c{\in}S_c \leftrightarrow \text{type}(c, \text{cat}) \wedge \exists m{\in}S_m.\, \exists f{\in}S_f(c, m)) \wedge \\ &\forall m(m{\in}S_m \leftrightarrow \text{type}(m, \text{mouse}) \wedge \exists c{\in}S_c.\, \exists f{\in}S_f(c, m)) \wedge \\ &\forall f.\, \forall c.\, \forall m(f{\in}S_f(c, m) \leftrightarrow \text{type}(f, \text{follow}) \wedge \text{arg1}(f, c) \wedge \text{arg2}(f, m)) \wedge \\ &\exists c{\in}S_c.\, \exists m{\in}S_m.\, \exists f{\in}S_f(c, m)).\end{aligned}$$

Note that in both examples, that last conjunct in the translation declares that the sets are non-empty. Without this conjunct, the first-order sentences are satisfied if all sets are empty, and so they could be true even if there are no cats, mice, or birds.

The general syntax of this fragment is given by the following Backus-Naur-like form:

$$A ::= c \mid x : f_1(A_1)\ \ldots\ f_n(A_n)$$

where c is a constant, x a variable, and f_i are elementary predicates, such as `type`, `arg1`, `arg1_of`, `arg2`, `arg2_of`, etc. To define its semantics, we define a function that recursively translates a logical form into first-order logic. This is necessary if we wish to use the well-developed reasoning and proof tools available for first-order logic. The defineability of this function also demonstrates that the language is unambiguous with respect to first-order logic: each logical form has a unique interpretation in first-order logic. For space, we leave the details of this translation function to section C.1.

It is straightforward to observe that the graph representation in this formalism is tree-structured, since there are no recurrent edges. Note that in terms of expressive power, this formalism is equivalent to Abstract Meaning Representation (AMR) without recurrent references or negation [1] and to Economical Discourse Representation Theory (EDRT) [3]. EDRT is shown to be in the two-variable fragment of first-order logic [3, 4] which is decidable, and as such, this simple formalism is decidable as well. In contrast, general first-order logic is undecidable.

4.2.2 Recurrent variable references

We can easily extend the above formalism to include recurrent variable references, in order to correctly represent sentences such as the following: This covers a wide range of anaphora phenomena within

Natural language	**Logical form**
"a bird likes flying"	`l:type(like) arg1(b:type(bird)) arg2(f:type(fly) arg1(b))`
"a cat grooms itself"	`g:type(groom) arg1(c:type(cat)) arg2(c)`

sentences (but certainly not exhaustively). The syntax is a simple extension of that of the above fragment:

$$A ::= c \mid x \mid x : f_1(A_1)\ \ldots\ f_n(A_n)$$

with the only difference being that the expression can be a reference to a variable x. The semantics are identical to that of the earlier formalism. However, the graph representation of logical forms in this formalism are no longer necessarily tree-structured. The translation of "A cat grooms itself" would be:

$$\begin{aligned}\exists S_g \exists S_c(&\forall c(c{\in}S_c \leftrightarrow \text{type}(c, \text{cat}) \wedge \exists g{\in}S_g(c)) \wedge \\ &\forall g.\, \forall c(g{\in}S_g(c) \leftrightarrow \text{type}(g, \text{groom}) \wedge \text{arg1}(g, c) \wedge \text{arg2}(g, c)) \wedge \\ &\exists c{\in}S_c.\, \exists g{\in}S_g(c)).\end{aligned}$$

This fragment is equivalent to AMR without negation (if polarity is translated to negation, as in [4]).

4.2.3 Universal quantification

Universal quantification is critically important in natural language, as it appears in a very large number of sentences in spoken and written language. It is also equally important in the formal language underlying the theory K in the reasoning module. Without universal quantification in K, the architecture is unable to generalize. As such, we present an extension to our logical formalism given above.

$$A ::= c \mid x \mid x[I_1, \ldots, I_k] : f_1(A_1)\ \ldots\ f_n(A_n)$$
$$I ::= x_1 \ldots x_n$$

We call the variables x_i in $I_1, \ldots, I_k$ *index variables*, and the variable x in $x[I_1, \ldots, I_k]$ is called *universally-quantified*.

Universal quantification adds new potential ambiguities with respect to quantifier scope. Consider the sentence "every cat likes every dog." In the *multiplicative reading* or *strong reading*, each cat likes every dog. In the *weak reading*, each cat likes at least one dog and each dog is liked by at least one cat. The logical forms corresponding to each reading for this sentence is given below: Note that if the quantifiers

Multiplicative reading: `l[c,d]:type(like) arg1(c:type(cat)) arg2(d:type(dog))`
Weak reading: `l[c d]:type(like) arg1(c:type(cat)) arg2(d:type(dog))`

are unbounded, as in this example, the multiplicative and strong readings are indistinguishable. The two

become distinct once bounded quantifiers are added in section 4.2.4. The first-order translations of the above readings are: Note that in the multiplicative/strong reading, there are four conjuncts: the first

Multiplicative/strong reading: $\exists S_l.\, \exists S_c.\, \exists S_d (\forall c (c \in S_c \leftrightarrow \text{type}(c, \text{cat})) \wedge \forall d (d \in S_d \leftrightarrow \text{type}(d, \text{dog})) \wedge \forall l.\, \forall c.\, \forall d (l \in S_l(c,d) \leftrightarrow \text{type}(l, \text{like}) \wedge \text{arg1}(l,c) \wedge \text{arg2}(l,d)) \wedge \forall c \in S_c.\, \forall d \in S_d.\, \exists l \in S_l(c,d))$.

Weak reading: $\exists S_l.\, \exists S_c.\, \exists S_d (\forall c (c \in S_c \leftrightarrow \text{type}(c, \text{cat})) \wedge \forall d (d \in S_d \leftrightarrow \text{type}(d, \text{dog})) \wedge \forall l.\, \forall c.\, \forall d (l \in S_l(c,d) \leftrightarrow \text{type}(l, \text{like}) \wedge \text{arg1}(l,c) \wedge \text{arg2}(l,d)) \wedge \forall c \in S_c.\, \exists d \in S_d.\, \exists l \in S_l(c,d) \wedge \forall d \in S_d.\, \exists c \in S_c.\, \exists l \in S_l(c,d))$.

defines S_c as the set of all cats, the second defines S_d as the set of all dogs, the third defines $S_l(c,d)$ as the set of all "like" events where c likes d, and the last conjunct posits that for every cat $c \in S_c$ and for every dog $d \in S_d$, there is a like event where c likes d. The weak reading is similar, except the last conjunct is divided into two: the first stating that for every cat $c \in S_c$, there exists a dog $d \in S_d$ where c likes d, and the second conjunct stating that for every dog $d \in S_d$, there exists a cat $c \in S_c$ where c likes d. We can use a short-hand to write these two conjuncts as $\forall^w c\, d \in S_c S_d.\, \exists l \in S_l(c,d)$ where we introduce a new $\forall^w$ operator denoting weak universal quantification:

$$\begin{aligned}\forall^w x_1 \ldots x_n \in S_1 \ldots S_n\ f(x_1, \ldots, x_n) \triangleq\ & \forall x_1 \in S_1 \exists x_2 \in S_2 \ldots \exists x_n \in S_n f(x_1, \ldots, x_n) \\ & \wedge \forall x_2 \in S_2 \underbrace{\exists x_1 \in S_1 \ldots \exists x_n \in S_n}_{\text{all } x_i \text{ except } x_2} f(x_1, \ldots, x_n) \\ & \wedge \ldots \wedge \forall x_n \in S_n \exists x_1 \in S_1 \ldots \exists x_{n-1} \in S_{n-1} f(x_1, \ldots, x_n).\end{aligned}$$

Scope trees: To properly describe the translation from this formalism to first-order logic, we need to formalize the notion of scope. We describe the scope of a logical form using a tree structure that we call a *scope tree*. Each vertex in this tree corresponds to a scope in the logical form, and is labeled with a (possibly empty) set of existentially-quantified variables. Every edge is labeled with exactly one universally-quantified variable $\forall x$ and an indicator that specifies whether the child scope is in the *antecedent* or the *consequent* of this universal quantification. Every non-index variable in the logical form must appear exactly once in the tree. Every index variable must appear exactly twice: once with the antecedent indicator, and once with the consequent indicator. The antecedent edge for a universally-quantified variable cannot be an ancestor or descendant of the corresponding consequent edge. See example in figure 4.3.

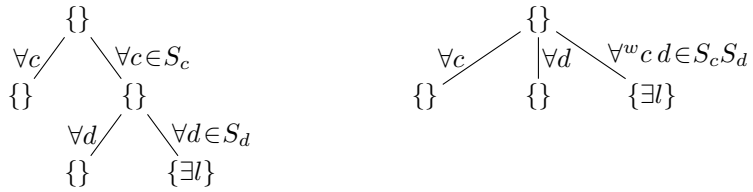


FIGURE 4.3: Scope trees for the multiplicative **(left)** and weak **(right)** readings of "every cat likes every dog."

See section C.3 for the general algorithm to compute the scope tree for a given logical form, and an algorithm to translate any logical form into first-order logic.

4.2.4 Raising and bounded universal quantification

Consider the sentence "everyone built a house." There are two possible interpretations: (1) everyone was involved in the construction of a single home, or (2) every person each constructed a house, and so there is one house per person. This sentence contains a universal quantifier over people and an existential quantifier over a house. The ambiguity boils down to the order of these two quantifiers. In (1), the existential quantifier is outside the universal quantifier (it takes broad scope), whereas in (2), the existential quantifier is inside the universal quantifier (it takes narrow scope). It is impossible to represent (1) with the formalism of the previous section. But we can add an operator, which we call *raising* and can be applied to any variable. This operator, written as a carat symbol `^`, moves the variable from one scope to its parent scope, and can be applied multiple times. Figure 4.4 depicts the logical forms and corresponding scope trees for the above two interpretations.

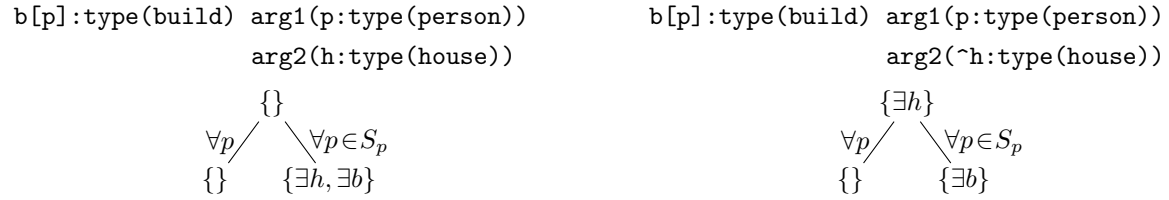


FIGURE 4.4: Logical forms and scope trees for the sentence "everyone built a house."

The syntax for this formalism is a simple extension to that of the previous section:

$A ::= c \mid x \mid V[I_1, \ldots, I_k] : f_1(A_1) \ \ldots \ f_n(A_n)$

$I ::= x_1 \ldots x_n \qquad V ::= x \mid \hat{}V$

The formalism in the previous section enables *unbounded* quantification, meaning quantification over the set of all objects that satisfy some property. However, there are many cases in natural language where universal quantification is *bounded*: i.e. over a subset of the objects that satisfy the property. For example, consider the sentence "Two bakers made three cakes." In the *weak reading*, there is a set of two bakers, and a set of three cakes, and each baker helped to make at least one cake. In the *strong reading*, again, we have a set of two bakers and a set of three cakes, and each baker helped to make *every* one of the three cakes. In the *multiplicative reading*, we have a set of two bakers, but now for each baker, there is a set of three cakes that the baker made, and so there may be up to *six* distinct cakes in total. To correctly represent the meaning of this sentence, we need a formalism in which sets are first-class citizens of the language. And to achieve this, we add the elementary predicates `set` and `size` (which specifies the size of a set). Variables with a `set` predicate are called *set variables*. A variable cannot be labeled with more than one `set` predicate. We do not allow set variables to be universally-quantified, but they are allowed to be index variables for other universally-quantified variables. The logical form expressions for each reading of this sentence are given: In these examples, the variable B represents a set

Multiplicative reading:	`m[b,c]:type(make)`	`arg1(B:set(b:type(baker)) size(2))` `arg2(C:set(c:type(cake)) size(3))`
Strong reading:	`m[b,c]:type(make)`	`arg1(B:set(b:type(baker)) size(2))` `arg2(^C:set(c:type(cake)) size(3))`
Weak reading:	`m[b c]:type(make)`	`arg1(B:set(b:type(baker)) size(2))` `arg2(C:set(c:type(cake)) size(3))`

of 2 bakers, whereas C represents a set of 3 cakes. The `set` predicate indicates that B contains objects that satisfy the constraints on b (i.e. $B \subseteq S_b$). Figure 4.5 demonstrates that raising the variable C in the multiplicative reading yields the strong reading of the sentence. Note that the antecedent scope for C (whose edge is labeled $\forall c$) is also moved. See section C.4 for details on how to compute the scope tree and the first-order translation.

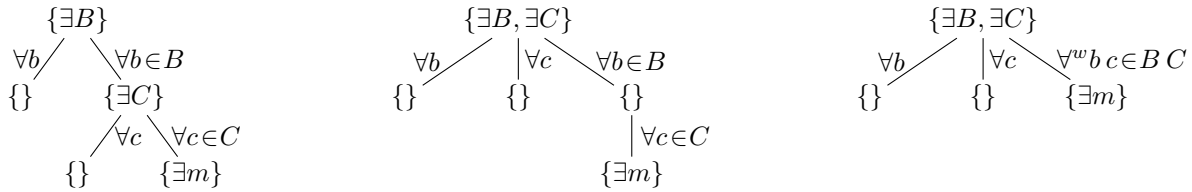


FIGURE 4.5: Scope trees for the multiplicative reading **(left)**, strong reading **(center)**, and weak reading **(right)** of "two bakers made three cakes."

4.2.5 Negation

Natural language very often contains words that indicate logical negation, and thus our logical formalism should be able to express negation. We add an operator, written as a minus `-`, to indicate negation.

$A ::= c \mid x \mid -A \mid V[I_1, \ldots, I_k] : f_1(A_1) \ \ldots \ f_n(A_n)$

$I ::= x_1 \ldots x_n \mid -I \qquad V ::= x \mid \hat{}V \mid -V$

Note that in the above grammar, ε is the empty symbol. Introducing negation also introduces new ambiguities. Consider the sentence "everyone didn't build a house." There are a very large number of possible interpretations: (1) there exists a person that did not build a house, (2) there is no one who built a house, (3) there does not exist a single house that everyone built, (4) they built something other than a house, and many more. We distinguish between two forms of negation: the first is the negation of constant symbols (when a negation precedes c in the above grammar), and the second is the negation of variables (all other occurences of `-` in the grammar). Negation of a variable x is equivalent to placing a $\neg$ operator in front of the corresponding quantifier for x, and this creates a new scope in the scope tree: Consider the formula $\neg\forall y.\, \exists x.\, f(x, y)$. Raising the x variable once produces $\neg\exists x.\, \forall y.\, f(x, y)$, and we can raise it once more to produce $\exists x.\, \neg\forall y.\, f(x, y)$. Negation of a constant symbol negates only the predicate instance in which that symbol appears, and does not alter the scope tree. Figure 4.6 provides examples of both kinds of negation for three (out of many more) interpretations.

Index variables cannot be negated where they are declared in the logical form. They must be negated when they are used as an index to quantify over another variable (for example, in figure 4.6, `p` may only be negated by writing `b[-p]` since `-p:type(person)` is not allowed).

See section C.5 for details on how to compute the scope tree and the first-order translation algorithm.

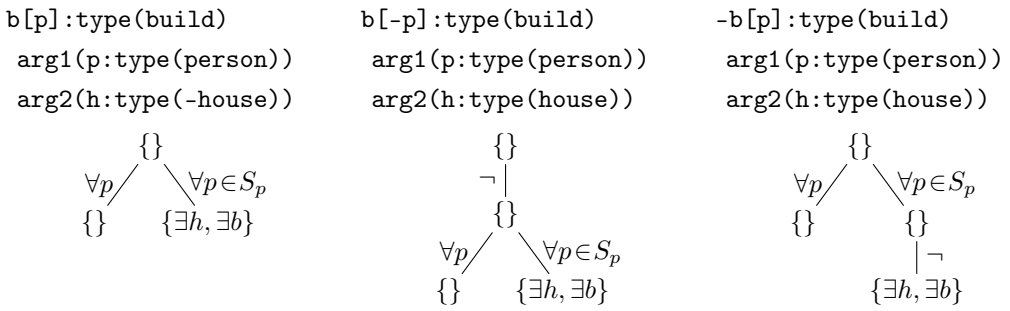


FIGURE 4.6: Logical forms and corresponding scope trees for the sentence "everyone didn't built a house." **(left)** Everyone built something other than a house. The constant symbol `house` is negated, and the scope tree is unchanged. **(center)** This negates the entire logical form since the quantifier $\forall p$ is at the root and the negation is placed in front of it. Thus, this interpretation is that there exists someone who did not build a house. **(right)** The negation is placed in front of $\exists b$. Since h is a child node of b, $\exists h$ is also in the scope of the negation. This is interpreted as everyone was involved in some activity that was not building a house.

4.3 Executive module

The executive module's purpose is active data collection: find a sequence of documents and sentences with the goal of maximizing performance on a pre-defined evaluation task. The ideal design for the executive module is highly dependent on the evaluation task. One architecture may perform well for one evaluation task, but perform poorly for a different task. We propose a simple algorithm for the executive module specifically for the question-answering task. At the start of execution, the algorithm is given the set of evaluation questions. For each question:

1. Parse the question, as well as the multiple choice options, if any are available. If the parser takes too long, return failure (skipping the question). If the parser returns a derivation tree containing unrecognized tokens, search for articles that define those tokens, such as from Simple English Wikipedia, Wiktionary, or a general-purpose search engine such as Google.
2. Parse the sentences in the article until the desired knowledge is acquired. For example, if we only need the definition of a term, then we only need to parse the definitional sentence. If the parser encounters a sentence that takes too long to parse, skip it and move on to the next sentence in the article. If the parser encounters unrecognized tokens, then the executive module again searches for articles that define those tokens, and we recursively repeat step 2. However, we impose a pre-defined limit on this recursion. If this limit is exceeded, the algorithm skips the sentence.
3. Once every token in the evaluation question is recognizeable to the language module, we attempt to answer the question with the current theory in the reasoning module. If the information in the theory is insufficient to answer the question, the algorithm searches for articles on the terms in the question, recursively performing step 2 again.

After going through the questions, it iterates over the questions again. If the algorithm fails and skips a question, it may be able to parse it in the second pass. Eventually, the reasoning and language modules will no longer improve with additional passes over the questions. At that point, the time limit on the parser or the recursion depth limit may be increased to allow the algorithm to explore a broader range of articles/sentences and spend more time searching for logical form derivations during parsing.

There is room to explore for more sophisticated active data collection algorithms. For example, if a general search engine is used, an appropriate search query must be generated in order to find the most relevant (and factual) articles. In addition, if the reasoning module is unable to answer a question, it would be more helpful if it could output exactly what information is missing, enabling a more targeted search for missing knowledge. In addition, evaluation tasks other than question-answering might require a different strategy for active data collection. For example, if the evaluation measures the coverage of the knowledge base, a breadth-first strategy may be more appropriate.

4.4 Proposed timeline

I will start working on the reasoning module since much of the thesis depends on it, with the goal of quickly obtaining a simple end-to-end system. I aim to first test the reasoning module with a hand-crafted set of logical forms $\{x_i\}$, and judge its ability to induct a reasonable theory K that logically entails $\{x_i\}$. We will start with the simplest formalism and build on it as needed. I aim to complete this by the end of the summer. Afterwards, I will write the production rules for a semantic grammar that will enable the language module to parse sentences into our logical formalism. With this done, I can test the language and reasoning modules end-to-end by the middle of the fall semester. I will work on a simple version of the executive module to test the end-to-end system on a hand-selected or constructed set of sentences. I will then broaden the complexity and expressivity of the logical forms $\{x_i\}$, working toward a set more representative of the facts expressed in declarative sentences, such as those in Simple

English Wikipedia. Finally, by next summer, I aim to complete the proposed work and evaluate on a dataset like Simple English Wikipedia.

Bibliography

[1] Laura Banarescu, Claire Bonial, Shu Cai, Madalina Georgescu, Kira Griffitt, Ulf Hermjakob, Kevin Knight, Philipp Koehn, Martha Palmer, and Nathan Schneider. Abstract meaning representation for sembanking. In *Proceedings of the 7th Linguistic Annotation Workshop and Interoperability with Discourse*, pages 178–186, Sofia, Bulgaria, August 2013. Association for Computational Linguistics.

[2] Phil Blunsom and Trevor Cohn. Inducing synchronous grammars with slice sampling. In *HLT-NAACL*, pages 238–241. The Association for Computational Linguistics, 2010. ISBN 978-1-932432-65-7.

[3] Johan Bos. Economical discourse representation theory. In Norbert E. Fuchs, editor, *CNL*, volume 5972 of *Lecture Notes in Computer Science*, pages 121–134. Springer, 2009. ISBN 978-3-642-14417-2.

[4] Johan Bos. Expressive power of abstract meaning representations. *Computational Linguistics*, 42 (3):527–535, 2016.

[5] Qingqing Cai and Alexander Yates. Semantic parsing freebase: Towards open-domain semantic parsing. In Mona T. Diab, Timothy Baldwin, and Marco Baroni, editors, **SEM@NAACL-HLT*, pages 328–338. Association for Computational Linguistics, 2013. ISBN 978-1-937284-48-0.

[6] Xinlei Chen, Abhinav Shrivastava, and Abhinav Gupta. Neil: Extracting visual knowledge from web data. In *ICCV*, pages 1409–1416. IEEE Computer Society, 2013. ISBN 978-1-4799-2839-2.

[7] Zhiyuan Chen and Bing Liu. *Lifelong Machine Learning*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2016.

[8] Noam Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2:113–124, 1956.

[9] Trevor Cohn, Phil Blunsom, and Sharon Goldwater. Inducing tree-substitution grammars. *Journal of Machine Learning Research*, 11:3053–3096, 2010.

[10] J. Cussens. Parameter estimation in stochastic logic programs. *Machine Learning Journal*, 44(3): 245–271, 2001.

[11] L. De Raedt and K. Kersting. Probabilistic inductive logic programming. In *Probabilistic Inductive Logic Programming*, pages 1–27. Springer, 2008.

[12] Li Dong and Mirella Lapata. Language to logical form with neural attention. *CoRR*, abs/1601.01280, 2016.

[13] Jenny Rose Finkel, Christopher D. Manning, and Andrew Y. Ng. Solving the problem of cascading errors: approximate bayesian inference for linguistic annotation pipelines. In *EMNLP '06: Proceedings of the 2006 Conference on Empirical Methods in Natural Language Processing*, pages 618–626, Morristown, NJ, USA, 2006. Association for Computational Linguistics.

[14] Alan E. Gelfand and Adrian F. M. Smith. Sampling-based approaches to calculating marginal densities. *Journal of the American Statistical Association*, 85(410):398–409, 1990.

[15] G. Gentzen. Untersuchungen über das logische schließen i. *Mathematische Zeitschrift*, 39:176–210, 1935.

[16] G. Gentzen. Investigations into Logical Deduction. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–213. North-Holland, Amsterdam, 1969.

[17] M. Johnson, T. L. Griffiths, and S. Goldwater. Bayesian inference for PCFGs via Markov chain Monte Carlo. In *Proceedings of the North American Conference on Computational Linguistics (NAACL '07)*, 2007.

[18] Tom Kwiatkowski, Luke S. Zettlemoyer, Sharon Goldwater, and Mark Steedman. Inducing probabilistic ccg grammars from logical form with higher-order unification. In *EMNLP*, pages 1223–1233. ACL, 2010. ISBN 978-1-932432-86-2.

[19] Tom Kwiatkowski, Luke S. Zettlemoyer, Sharon Goldwater, and Mark Steedman. Lexical generalization in ccg grammar induction for semantic parsing. In *EMNLP*, pages 1512–1523. ACL, 2011. ISBN 978-1-937284-11-4.

[20] Tom Kwiatkowski, Eunsol Choi, Yoav Artzi, and Luke S. Zettlemoyer. Scaling semantic parsers with on-the-fly ontology matching. In *EMNLP*, pages 1545–1556. ACL, 2013. ISBN 978-1-937284-97-8.

[21] Peng Li, Yang Liu, and Maosong Sun. An extended ghkm algorithm for inducing lambda-scfg. In Marie desJardins and Michael L. Littman, editors, *AAAI*. AAAI Press, 2013. ISBN 978-1-57735-615-8.

[22] Tom M. Mitchell, William Cohen, Estevam Hruschka, Partha Talukdar, Justin Betteridge, Andrew Carlson, Bhavana Dalvi, Matthew Gardner, Bryan Kisiel, Jayant Krishnamurthy, Ni Lao, Kathryn Mazaitis, Thahir Mohammad, Ndapa Nakashole, Emmanouil A. Platanios, Alan Ritter, Mehdi Samadi, Burr Settles, Richard Wang, Derry Wijaya, Abhinav Gupta, Xinlei Chen, Abulhair Saparov, Malcolm Greaves, and Joel Welling. Never-ending learning. In *AAAI*, 2015. : Never-Ending Learning in AAAI-2015.

[23] S. Muggleton. Inductive logic programming. *New Generation Computing*, 8(4):295–318, 1991.

[24] Stephen Muggleton. Stochastic logic programs. In *Advances in Inductive Logic Programming*, pages 254–264, 1996.

[25] Terence Parsons. *Events in the Semantics of English*. MIT Press, Cambridge, MA, 1990.

[26] Siva Reddy, Oscar Täckström, Slav Petrov, Mark Steedman, and Mirella Lapata. Universal semantic parsing. *CoRR*, abs/1702.03196, 2017.

[27] C. P. Robert and G. Casella. *Monte Carlo Statistical Methods*. Springer, New York, NY, 2010.

[28] Alexandre Rondon, Helena de Medeiros Caseli, and Carlos Ramisch. Never-ending multiword expressions learning. In Valia Kordoni, Kostadin Cholakov, Markus Egg, Stella Markantonatou, and Shuly Wintner, editors, *MWE@NAACL-HLT*, pages 45–53. The Association for Computer Linguistics, 2015. ISBN 978-1-941643-38-9.

[29] Abulhair Saparov, Vijay Saraswat, and Tom Mitchell. A probabilistic generative grammar for semantic parsing. In *Proceedings of the 21st Conference on Computational Natural Language Learning (CoNLL 2017)*, pages 248–259. Association for Computational Linguistics, August 2017.

[30] T. Sato, Y. Kameya, and N.-G. Zhou. Generative modeling with failure in PRISM. In F. Giunchiglia and L. Pack Kaelbling, editors, *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI-05)*, pages 847–852. AAAI Press, July 30,– August, 5 2005.

[31] Ashutosh Saxena, Ashesh Jain, Ozan Sener, Aditya Jami, Dipendra Kumar Misra, and Hema Swetha Koppula. Robobrain: Large-scale knowledge engine for robots. *CoRR*, abs/1412.0691, 2014.

[32] Shashank Srivastava, Igor Labutov, and Tom M. Mitchell. Joint concept learning and semantic parsing from natural language explanations. In Martha Palmer, Rebecca Hwa, and Sebastian Riedel, editors, *EMNLP*, pages 1527–1536. Association for Computational Linguistics, 2017. ISBN 978-1-945626-83-8.

[33] Yu Su and Xifeng Yan. Cross-domain semantic parsing via paraphrasing. *CoRR*, abs/1704.05974, 2017.

[34] Yee Whye Teh, Michael I. Jordan, Matthew J. Beal, and David M. Blei. Hierarchical dirichlet processes. *Journal of the American Statistical Association*, 101(476):1566–1581, 2006.

[35] Sebastian Thrun and Tom M. Mitchell. Lifelong robot learning. *Robotics and Autonomous Systems*, 15(1-2):25–46, 1995.

[36] Aaron Steven White, Drew Reisinger, Keisuke Sakaguchi, Tim Vieira, Sheng Zhang, Rachel Rudinger, Kyle Rawlins, and Benjamin Van Durme. Universal decompositional semantics on universal dependencies. In Jian Su, Xavier Carreras, and Kevin Duh, editors, *EMNLP*, pages 1713–1723. The Association for Computational Linguistics, 2016. ISBN 978-1-945626-25-8.

[37] Yuk Wah Wong and Raymond J. Mooney. Learning for semantic parsing with statistical machine translation. In Robert C. Moore, Jeff A. Bilmes, Jennifer Chu-Carroll, and Mark Sanderson, editors, *HLT-NAACL*. The Association for Computational Linguistics, 2006.

[38] Yuk Wah Wong and Raymond J. Mooney. Learning synchronous grammars for semantic parsing with lambda calculus. In John A. Carroll, Antal van den Bosch, and Annie Zaenen, editors, *ACL*. The Association for Computational Linguistics, 2007.

[39] Luke S. Zettlemoyer and Michael Collins. Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. In *UAI*, pages 658–666. AUAI Press, 2005. ISBN 0-9749039-1-4.

[40] Luke S. Zettlemoyer and Michael Collins. Online learning of relaxed ccg grammars for parsing to logical form. In Jason Eisner, editor, *EMNLP-CoNLL*, pages 678–687. ACL, 2007.

[41] Kai Zhao and Liang Huang. Type-driven incremental semantic parsing with polymorphism. *CoRR*, abs/1411.5379, 2014.

A. Natural deduction

This section provides a very brief introduction to natural deduction. An application of a deduction rule in natural deduction can be written as:

$$\frac{\phi_1 \quad \phi_2 \quad \dots \quad \phi_n}{\psi}\, d$$

where $d \in \mathcal{D}$ is the name of the deduction rule, $\phi_1, \dots, \phi_n$ are the premises of the deduction, and ψ is the conclusion. The deduction rules in natural deduction are shown in figure A.1.

conjunction introduction
$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \wedge\text{I}$$

conjunction elimination
$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} \wedge\text{E}_L \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B} \wedge\text{E}_R$$

disjunction introduction
$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \vee\text{I}_L \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B} \vee\text{I}_R$$

disjunction elimination
$$\frac{\Gamma \vdash A \vee B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C} \vee\text{E}$$

implication introduction
$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B} \to\text{I}$$

implication elimination
$$\frac{\Gamma \vdash A \to B \quad \Gamma \vdash A}{\Gamma \vdash B} \to\text{E}$$

proof by contradiction
$$\frac{\Gamma, \neg A \vdash \bot}{\Gamma \vdash A}\text{C}$$

negation elimination
$$\frac{\Gamma \vdash A \quad \Gamma \vdash \neg A}{\Gamma \vdash \bot} \neg\text{E}$$

truth introduction
$$\frac{}{\Gamma \vdash \top} \top\text{I}$$

falsehood elimination
$$\frac{\Gamma \vdash \bot}{\Gamma \vdash A} \bot\text{E}$$

universal introduction
$$\frac{\Gamma \vdash [a/x]A}{\Gamma \vdash \forall x.\, A} \forall\text{I}$$

universal elimination
$$\frac{\Gamma \vdash \forall x.\, A}{\Gamma \vdash [t/x]A} \forall\text{E}$$

existential introduction
$$\frac{\Gamma \vdash [t/x]A}{\Gamma \vdash \exists x.\, A} \exists\text{I}$$

existential elimination
$$\frac{\Gamma \vdash \exists A \quad \Gamma, [a/x]A \vdash C}{\Gamma \vdash C} \exists\text{E}$$

axiom
$$\frac{}{\Gamma, A \vdash A}\text{Ax}$$

substitution
$$\frac{\Gamma \vdash A \quad \Gamma', A \vdash B}{\Gamma, \Gamma' \vdash B}\text{Sub}$$

Figure A.1: Deduction rules in classical natural deduction. In each of these rules, Γ is a comma-separated set of logical forms. The notation $[a/x]A$ denotes a substitution of x for the term a in the logical form A.

Applications of natural deduction rules can be stacked upon each other, in which the conclusion of an earlier rule is used as a premise in a later rule. Thus, natural deduction proofs can be depicted as "trees," where the premises are higher in the tree. An example of the proof of $A \wedge B \vdash B \wedge A$ is shown:

$$\dfrac{\dfrac{\dfrac{}{A \wedge B \vdash A \wedge B}\text{Ax}}{A \wedge B \vdash B} \wedge\text{E}_R \quad \dfrac{\dfrac{}{A \wedge B \vdash A \wedge B}\text{Ax}}{A \wedge B \vdash A} \wedge\text{E}_L}{A \wedge B \vdash B \wedge A} \wedge\text{I}$$

B. Proposed proof transformations for Metropolis-Hastings

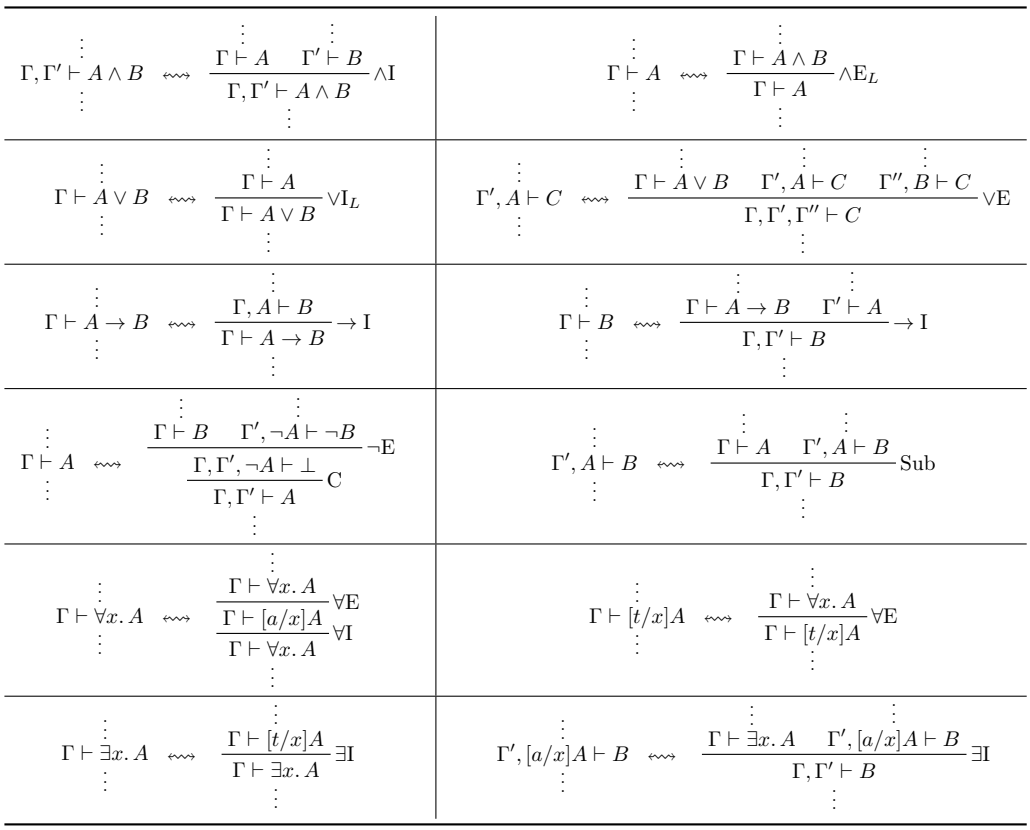


Figure B.1: A set of proposed local transformations for proofs in the Metropolis-Hastings algorithm. We omit a handful of transformations for brevity, such as $\wedge \mathrm{E}_R$ and $\vee \mathrm{I}_R$.

C. Semantic representation: Translation to first-order logic

C.1 Simple logical formalism

In this section, we detail the function that translates from logical forms in our simple formalism into first-order logic. To help define this function, we rely on a helper function $N^+(\cdot)$, which takes a variable x as input, and outputs the collection of neighboring vertices for which there is an outgoing edge from x. In the graph representation, $N^+(x)$ is the collection of neighboring vertices $(n_1, \ldots, n_k)$ of x for which there is a directed edge from x to n_i for all $i = 1, \ldots, k$. For example, in the logical form for "A cat follows a mouse" above, $N^+(f) = \{c, m\}$, whereas $N^+(c) = \{\}$. We define the first-order translation of

a logical form ϕ in algorithm 1.

Algorithm 1: Algorithm that translates a logical form ϕ in the simple formalism into a formula in first-order logic.

foreach *variable* x *in* ϕ *(for each vertex in the graph)* **do**
  emit existential quantifier for the set "$\exists S_x$"
foreach *variable* x *in* ϕ **do**
  emit a universal quantifier "$\forall x$"
  foreach *variable* $n_i \in N^+(x)$ **do** emit a universal quantifier "$\forall n_i$"
  emit "$x \in S_x(\boldsymbol{n})$" where $\boldsymbol{n} \triangleq (n_1, \ldots, n_k) = N^+(x)$
  emit the bidirectional symbol "$\leftrightarrow$"
  foreach *outgoing predicate edge* (x, y) *with predicate* f_i **do**
    emit a conjunct "$f_i(x, y)$"
    /* An edge is considered *outgoing* from x if either: (1) x is the source vertex of a predicate type, arg1, arg2, etc, or (2) x is the destination vertex of a predicate arg1_of, arg2_of, etc. In the latter case, we "invert" the predicate, and so arg1_of becomes arg1, for example. */
  foreach *incoming predicate edge* (y, x) *with predicate* f_i **do**
    emit a conjunct "$\exists \boldsymbol{u}^* \in S_{\boldsymbol{u}^*}. \exists y \in S_y(\boldsymbol{u})$"
    /* $\boldsymbol{u} \triangleq (u_1, \ldots, u_k) = N^+(y)$, $\boldsymbol{u}^*$ is the set of all vertices u from which there exists a directed path from y to u, not including x or y, and $S_{\boldsymbol{u}^*}$ is the set of set variables corresponding to the variables in $\boldsymbol{u}^*$. */
emit the final conjunct "$\exists \boldsymbol{v}^* \in S_{\boldsymbol{v}^*}. \exists r \in R(\boldsymbol{r})$"
/* r is the root of the logical form, S_r is the set variable corresponding to r, $\boldsymbol{v} \triangleq (v_1, \ldots, v_k) = N^+(r)$, $\boldsymbol{v}^*$ and $S_{\boldsymbol{v}^*}$ are defined similarly to $\boldsymbol{u}^*$ and $S_{\boldsymbol{u}^*}$ above. */

C.2 Recurrent variable references

The algorithm is the same as above. Note that $N^+(\cdot)$ includes recurrent edges in its definition.

C.3 Universal quantification

Computing the scope tree: To compute the scope tree of an arbitrary logical form ϕ, we execute the following recursive procedure, starting with the root variable in ϕ and an initial scope tree with an empty scope at the root {}:

Algorithm 2: Algorithm that constructs the scope tree for a logical form in the formalism with unbounded universal quantification.

function `make_scope_tree`(*variable* x, *node in scope tree* $\boldsymbol{s}$)
  if x *has index variables, as in* $x[I_1, \ldots, I_n]$
    $\mathbf{s}_0 \leftarrow \mathbf{s}$
    for $i \in \{1, \ldots, n\}$ **do**
      foreach $y \in I_i$ **do**
        add a child node $\mathbf{a}_y$ to $\mathbf{s}_{i-1}$ /* $\mathbf{a}_y$ is the antecedent scope */
        label its edge $\forall y$
      add a child node $\mathbf{s}_i$ to $\mathbf{s}_{i-1}$ /* $\mathbf{s}_i$ is the consequent scope */
      label its edge $\forall^w I_i \in S_{I_i}$
  if x *has index variables*
    $\mathbf{s} \leftarrow \mathbf{s}_n$
  if x *is not currently in the scope tree*
    add $\exists x$ to $\mathbf{s}$
  foreach *child* y *of* x **do**
    if y *is an index variable* call `make_scope_tree`$(y, \mathbf{a}_y)$
    else call `make_scope_tree`$(y, \mathbf{s})$

We define a function $\mathcal{A}(x)$ which will become useful in the translation to first-order logic. $\mathcal{A}$ takes as input a variable x. If x is an index variable, we find the edge labeled $\forall x$ in the scope tree and we let $\mathbf{n}$ be the parent node of this edge. If x is not an index variable, we find the node with the label $\exists x$ and we let this node be $\mathbf{n}$. Then from $\mathbf{n}$, we walk along the edges toward the root of the scope tree, collecting the variables in the edge labels. Thus, $\mathcal{A}(x) \triangleq \{y :$ there is an edge with label $\forall y$ or $\forall y \in S_y$ in the path from $\mathbf{n}$ to the root$\}$.

Further restrictions on logical forms: We restrict logical forms so that for any universally-quantified variable $x[I_1, \ldots, I_n]$, the index variables $x_i \in I_j$ must be distinct descendants of x (where "descendant" does not include recurrent reference edges). The edge connecting x to its parent must be outgoing from x to the parent. For any two weakly-quantified indices $x_a, x_b \in I_i$, x_a cannot be an ancestor or descendant of x_b. Finally, for any I_i and I_j where $i < j$, no variable in I_i may be a descendant of any variable in I_j.

In addition, if a logical form has multiple universally-quantified variables, they cannot "overlap." To be more precise, let $x[I_1, \ldots, I_n]$ and $x'[I'_1, \ldots, I'_n]$ be any two distinct universally-quantified variables in a logical form, and without loss of generality, let x be closer to the root. Then exactly one of the following (mutually exclusive) conditions must hold: (1) x is not an ancestor of x', or (2) x is an ancestor of x', and no index variable x_i in $I_1, \ldots, I_n$ is a descendant of x'.

Further, we make a restriction on recurrent reference edges. Given two scopes, $\mathbf{s}_1$ and $\mathbf{s}_2$, we say (inductively) that $\mathbf{s}_1$ is *referable from* $\mathbf{s}_2$ if either: (1) $\mathbf{s}_1 = \mathbf{s}_2$, (2) $\mathbf{s}_1$ is referable from the parent scope of $\mathbf{s}_2$, or (3) the consequent scope of $\mathbf{s}_1$ is referable from $\mathbf{s}_2$. So, given two variables, x and y, we say x is referable from y if the scope containing x is referable from the scope containing y. We only allow a recurrent reference edge from y to x if x is referable from y.

First-order translation: To provide a first-order translation, we need a *dependency graph* D that provides, for each variable x, a set of variables that x depends on. Thus, D is a simple directed graph where each vertex corresponds to a variable in the logical form. We can use any simple directed graph that satisfies the following properties: (1) x and y are connected by an edge in the logical form if and only if there is an edge between x and y in D (note we did not specify the edge direction), and (2) if $y \in \mathcal{A}(x)$, there cannot be an edge (y, x) in D. One simple way to construct D is to start with the graph representation of the original logical form and simply flip the directions of edges that violate the second property. As convenient notation, let $N^+(x) \triangleq \{y : (x, y) \in E(D)\}$ be the neighbors of x in D connected by an outgoing edge from x, and let $N^-(x) \triangleq \{y : (y, x) \in E(D)\}$ be the neighbors of x connected by an incoming edge to x. Given the scope tree and the dependency graph D, we can translate a logical form ϕ into first-order logic using algorithm 3.

Algorithm 3: Algorithm that translates a logical form ϕ in the formalism with universal quantification into a formula in first-order logic.

1 **foreach** *variable* x *in* ϕ *(for each vertex in the graph)* **do**
2 emit existential quantifier for the set "$\exists S_x$"
3 **foreach** *variable* x *in* ϕ **do**
4 emit a universal quantifier "$\forall x$"
5 **foreach** *variable* $n_i \in N^+(x) \cup \mathcal{A}(x)$ **do** emit a universal quantifier "$\forall n_i$"
6 emit "$x \in S_x(\boldsymbol{n})$" where $\boldsymbol{n} \triangleq (n_1, \ldots, n_k) = N^+(x) \cup \mathcal{A}(x)$
7 emit the bidirectional symbol "$\leftrightarrow$"
8 **foreach** *edge from* x *to a constant symbol* c *with predicate* f_i **do** emit conjunct "$f_i(x, c)$"
9 **foreach** *variable* $y \in N^+(x)$ *with predicate* f_i **do**
10 emit a conjunct "$f_i(x, y)$"
11 **foreach** *variable* $y \in N^-(x)$ *with predicate* f_i **do**
12 **if** x *is a descendant of an antecedent scope and* y *is not a descendant of that scope*
13 **continue**
 `/* reaching this point implies that` x `is either in the same scope as` y `or in an ancestor scope of` y `*/`
14 emit a conjunct "$\exists \boldsymbol{u}^* \in S_{\boldsymbol{u}^*}. \exists y \in S_y(\boldsymbol{u})$"
15 **foreach** *scope node* $\boldsymbol{s}$ *along the path in the scope tree from* x *down to* y **do**
16 **if** $\boldsymbol{s}$ *has a parent edge along this path* emit the label on the edge
17 **foreach** *variable* u *in* $\boldsymbol{s}$ *such that there exists a directed path from* y *to* u **do**
18 emit existential quantifier "$\exists u \in S_u(N^+(u) \cup \mathcal{A}(u))$"
19 emit "$\exists y \in S_y(N^+(y) \cup \mathcal{A}(y))$".
20 emit the final conjunct, using the same procedure as in line 15, starting from the root scope of the scope tree, and traversing the tree along the path to the scope containing r, the root variable of the logical form, finally emitting "$\exists r \in S_r(N^+(r) \cup \mathcal{A}(r))$" at the end.

C.4 Raising and bounded universal quantification

The algorithm to compute the scope tree is largely similar to that in the previous section. We execute the following recursive procedure, starting with the root variable and an initial scope tree with an empty

scope at the root {}:

Algorithm 4: Algorithm that constructs the scope tree for a logical form in the formalism with bounded universal quantification and raising.

1 **function** `make_scope_tree(`*variable* x*, node in scope tree* $\mathbf{s}$`)`
2 **foreach** *raising operator* ^ *on* x **do**
3 $\mathbf{s} \leftarrow$ parent of $\mathbf{s}$
4 **if** x *has index variables, as in* $x[I_1, \ldots, I_n]$
5 $\mathbf{s}_0 \leftarrow \mathbf{s}$
6 **for** $i \in \{1, \ldots, n\}$ **do**
7 **foreach** $y \in I_i$ **do**
8 add a child node $\mathbf{a}_y$ to $\mathbf{s}_{i-1}$ `/* a`$_y$ `is the antecedent scope */`
9 label its edge $\forall y$
10 add a child node $\mathbf{s}_i$ to $\mathbf{s}_{i-1}$ `/* s`$_i$ `is the consequent scope */`
11 label its edge $\forall^w I_i \in \tilde{S}(I_i)$
`/* where` $\tilde{S}(x_1 \ldots x_{k-1}x_k) \triangleq \tilde{S}(x_1 \ldots x_{k-1})X_k$ `if the parent of` x_k `is a set variable` X_k`, or` $\tilde{S}(x_1 \ldots x_{k-1}x_k) \triangleq \tilde{S}(x_1 \ldots x_{k-1})S_{x_k}$ `otherwise */`
12 **if** x *has index variables*
13 $\mathbf{s} \leftarrow \mathbf{s}_n$
14 **if** x *is not currently in the scope tree*
15 add $\exists x$ to $\mathbf{s}$
16 **foreach** *child* y *of* x **do**
17 **if** y *is an index variable* call `make_scope_tree`$(y, \mathbf{a}_y)$
18 **elif** y *is a set variable for an index variable* z *(the variable in* y*'s* `set` *predicate is the index variable* z*)*
19 call `make_scope_tree`$(y, \mathbf{s}_{i-1})$ where $z \in I_i$
20 **elif** x *is a set variable for* y *and* y *is not an index variable*
21 create child scope node $\mathbf{c}_y$ of $\mathbf{s}$
22 label its edge $\forall y \in x$
23 call `make_scope_tree`$(y, \mathbf{c}_y)$
24 **else** call `make_scope_tree`$(y, \mathbf{s})$

Given the scope tree, the first-order translation is identical to algorithm 3 except we add another constraint to the dependency graph D: (3) if an edge (x, y) is labeled with the predicate `set` in the logical form, then (x, y) is not an edge in D. Whenever the translation algorithm encounters a variable x with a predicate `set` with the variable y, we emit "$x \subseteq S_y$", or equivalently "$\forall z(z \in x \rightarrow z \in S_y)$", in its place.

C.5 Negation

The algorithm to construct the scope tree for a given logical form is almost identical to algorithm 4. We add the following step before line 2: If the current variable has a negation operator proceding it, subdivide the parent edge of the current node in the scope tree, splitting the edge into two edges with a new empty node in between. The edge closer to the root retains the label of the old edge, and we label the other new edge ¬. Note that if the current node has no parent, we create it and label the connecting edge ¬. Note that if there are multiple negation operators, we repeat this process as needed. We also modify line 4: if I_i has a negation operator, perform the same subdivision operation described above on the node $\mathbf{s}_{i-1}$.

The first-order translation algorithm is also almost identical to algorithm 3. One difference is that we add a fourth requirement for the dependency graph D: (4) for any two variables x and y, if the path from x to y in the scope tree traverses an edge (u, v) labeled ¬ and u is the parent of v, then the edge (x, y) cannot be in D. Also, whenever the algorithm encounters an edge with predicate f_i connecting the variable x and a constant symbol c, we now check if c is negated. If so, we instead emit the conjunct "$\neg f_i(x, c)$".